

ПРАКТИЧЕСКАЯ РАБОТА №3. УСЛОВНАЯ ЛОГИКА, ПОДЗАПРОСЫ И ОБОБЩЕННЫЕ ТАБЛИЧНЫЕ ВЫРАЖЕНИЯ (СТЕ) В POSTGRES PRO

Цель работы:

Работа направлена на формирование глубокого понимания и практического применения инструментов для реализации сложной бизнес-логики непосредственно на уровне базы данных.

По завершении работы студент должен уметь:

- Реализовывать сложную условную логику внутри SQL-запросов с использованием выражения CASE для категоризации данных, обработки исключительных ситуаций и динамического вычисления значений.
- Сформировать концептуальное понимание подзапросов, их классификации (скалярные, многострочные, коррелированные) и практического применения для выполнения динамической фильтрации и вычислений, основанных на результатах других запросов.
- Освоить синтаксис и методологию использования обобщенных табличных выражений (СТЕ) для декомпозиции сложных запросов, повышения их читаемости, структурированности и поддерживаемости.
- Приобрести навыки работы с иерархическими и древовидными структурами данных, научившись составлять рекурсивные запросы с помощью WITH RECURSIVE для обхода и анализа таких структур.

Постановка задачи:

Задание 1: использование оператора CASE

1. Составить запрос, использующий поисковое выражение CASE для категоризации данных по какому-либо числовому признаку из вашей БД (например, цена, количество, возраст). Запрос должен содержать **не менее трех условий WHEN** и ветку **ELSE**.
2. Составить запрос, в котором оператор CASE используется внутри **агрегатной функции** (например, SUM или COUNT) для выполнения условной агрегации.

Задание 2: использование подзапросов.

Составить и выполнить три запроса, демонстрирующих разные типы подзапросов.

1. **Скалярный подзапрос:** найти все записи в таблице, у которых значение в некотором числовом столбце превышает среднее (или максимальное/минимальное) значение по этому столбцу.
2. **Многострочный подзапрос с IN:** вывести информацию из одной таблицы на основе идентификаторов, полученных из связанной таблицы по определенному критерию (в данном случае, **обязательно по дате**).
3. **Коррелированный подзапрос с EXISTS:** найти все записи из родительской таблицы, для которых существует хотя бы одна связанная запись в дочерней таблице, удовлетворяющая текстовому условию.
4. **Альтернативное решение с JOIN:** решите задачу из пункта выше (2.3, Коррелированный подзапрос с EXISTS), но на этот раз с использованием оператора соединения **JOIN**.

Задание 3: использование обобщенных табличных выражений (СТЕ).

1. **Стандартное СТЕ:** переписать запрос из Задания 2.3 (с коррелированным подзапросом) с использованием обобщенного табличного выражения (СТЕ).
2. **Рекурсивное СТЕ:** используя имеющуюся в вашей схеме данных таблицу с иерархической структурой (например, pharmacists), написать рекурсивный запрос с помощью **WITH RECURSIVE** для вывода всей иерархии с указанием уровня вложенности.

Примечание: если в вашей схеме данных отсутствует таблица с иерархической структурой (т.е. таблица, которая ссылается сама на себя), вам необходимо создать демонстрационную таблицу для выполнения этого задания.

Вы можете выбрать один из двух подходов:

1. **Модифицировать существующую таблицу:** если у вас есть таблица employees, staff или подобная, Вы можете добавить в неё столбец manager_id INT и внешний ключ, ссылающийся на первичный ключ этой же таблицы.
2. **Создать новую таблицу:** создайте простую таблицу для демонстрации иерархии, например, для категорий товаров.

Листинг 0. Пример создания и наполнения таблицы для категорий

```
CREATE TABLE categories (  
    category_id SERIAL PRIMARY KEY,  
    name VARCHAR(255) NOT NULL,  
    parent_id INT, -- Ссылка на родительскую категорию  
    CONSTRAINT fk_parent_category  
        FOREIGN KEY(parent_id)  
        REFERENCES categories(category_id)  
        ON DELETE CASCADE  
);  
  
INSERT INTO categories (category_id, name, parent_id) VALUES  
(1, 'Электроника', NULL),  
(2, 'Бытовая техника', NULL),  
(3, 'Смартфоны', 1),  
(4, 'Ноутбуки', 1),  
(5, 'Холодильники', 2),  
(6, 'Аксессуары для смартфонов', 3);
```

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

Введение

В предыдущих работах акцент делался на создании структуры базы данных и выполнении запросов к отдельным таблицам или их простым соединениям.

Эта работа посвящена более сложным аналитическим инструментам SQL, которые позволяют реализовывать комплексную бизнес-логику, строить динамические условия и работать со сложными структурами данных непосредственно на уровне СУБД.

Ниже приводятся таблицы, используемые для построения запросов.

Таблица 1. Таблица *manufacturers* (Производители)

<small>123</small> <small>↕</small> manufacturer_id	<small>A-Z</small> manufacturer_name	<small>A-Z</small> country
1	ООО "Фармстандарт"	Россия
2	Bayer AG	Германия
3	Sanofi	Франция

Таблица 2. Таблица *medicines* (Лекарства)

<small>123</small> <small>↕</small> id	<small>A-Z</small> name	<small>123</small> quantity_in_stock	<small>123</small> price	<small>🕒</small> production_date	<small>🕒</small> expiration_date	<small>123</small> <small>↕</small> manufacturer_id
1	Парацетамол	200	50,5	2025-07-10	2028-07-10	1
2	Аспирин	150	120	2025-07-12	2027-07-12	2
3	Ибупрофен	100	85	2025-07-11	2028-07-11	1
4	Витамин С	300	250	2025-07-10	2027-07-10	2
5	Активированный уголь	500	25	2025-07-10	2030-07-10	1

Таблица 3. Таблица *customers* (Покупатели)

<small>123</small> <small>↕</small> customer_id	<small>A-Z</small> first_name	<small>A-Z</small> last_name	<small>A-Z</small> email	<small>A-Z</small> phone_number
1	Иван	Иванов	ivan@example.com	+79001234567
2	Петр	Петров	petr@example.com	[NULL]

Таблица 4. Таблица *pharmacists* (Фармацевты)

<small>123</small> <small>↕</small> pharmacist_id	<small>A-Z</small> first_name	<small>A-Z</small> last_name	<small>A-Z</small> phone_number	<small>123</small> <small>↕</small> manager_id
1	Иван	Сорокин	+7(905)123-44-56	[NULL]
2	Алексей	Петров	+7(905)123-44-56	1
3	Марина	Иванова	+7(905)123-44-56	1
4	Евгений	Смирнов	+7(905)123-44-56	2
5	Александр	Орлов	+7(905)123-44-56	3

Таблица 5. Таблица sales (Продажи)

123 sale_id ▼	123 customer_id ▼	🕒 sale_date ▼	123 total_amount ▼
1	1	2025-01-22	221
2	[NULL]	2025-01-23	1 555,5

Таблица 6. Таблица sale_items (Позиции продаж)

123 sale_item_id ▼	123 sale_id ▼	123 medicine_id ▼	123 quantity ▼	123 unit_price ▼
1	1	1	2	50,5
2	1	2	1	120
3	2	3	3	85
4	2	1	1	50,5
5	2	4	5	250

1. Реализация условной логики – оператор CASE

Оператор **CASE** – это аналог конструкции **if-then-else** из языков программирования, позволяющий выполнять условную логику прямо в SQL-запросе.

Он вычисляет список условий и возвращает одно из нескольких возможных выражений. Это мощный инструмент для категоризации данных и реализации сложных бизнес-правил.

1.1 Категоризация данных

Одной из самых частых задач в аналитике является категоризация данных – присвоение записям определенных **меток** или **групп** на основе их **атрибутов**.

Например, вместо того чтобы смотреть на точную цену товара, аналитику может быть удобнее оперировать понятиями «дешевый», «средний» и «дорогой».

Оператор CASE идеально подходит для этой задачи, позволяя «на лету» создавать новые столбцы с такими категориями.

Листинг 1. Категоризация лекарств по цене

```
-- Вывести название, цену и ценовую категорию для каждого лекарства
SELECT
    name, price,
    CASE
        WHEN price > 200 THEN 'Дорогое'
        WHEN price BETWEEN 80 AND 200 THEN 'Средняя цена'
        WHEN price < 80 AND price > 0 THEN 'Дешевое'
        ELSE 'Цена не указана'
    END AS price_category
FROM
    medicines;
```

Этот запрос для каждой строки таблицы medicines последовательно проверяет условия в блоке CASE.

Для каждого лекарства, цена (price) которого больше 200, в новом столбце price_category будет значение 'Дорогое'.

Если это условие ложно, проверяется следующее (BETWEEN 80 AND 200), и так далее.

Если ни одно из условий WHEN не выполнилось, будет использовано значение из блока ELSE.

	A-Z name	price	A-Z price_category
1	Парацетамол	50,5	Дешевое
2	Аспирин	120	Средняя цена
3	Ибупрофен	85	Средняя цена
4	Витамин С	250	Дорогое
5	Активированн	25	Дешевое

Рисунок 1 – Результат запроса с использованием CASE

1.2 Условная агрегация

Условная агрегация – это техника, позволяющая вычислять агрегатные функции (такие как COUNT, SUM, AVG) не по всей группе записей, а только по тем из них, которые **удовлетворяют определенному условию**.

Это достигается путем помещения оператора **CASE** внутрь агрегатной функции.

Такой подход позволяет в одном запросе получить несколько разных метрик для одной и той же группы, избегая сложных подзапросов.

Листинг 2. Подсчет количества лекарств по категориям запасов

```
-- Для каждого производителя посчитать, сколько у него "дефицитных"
-- (менее 150 шт.) и "избыточных" (более 150 шт.) лекарств.
SELECT
    m.manufacturer_name,
    COUNT(
        CASE
            WHEN med.quantity_in_stock < 150 THEN 1
            ELSE NULL
        END
    ) AS low_stock_medicines,
    COUNT(
        CASE
            WHEN med.quantity_in_stock >= 150 THEN 1
            ELSE NULL
        END
    ) AS high_stock_medicines
FROM
    manufacturers AS m
JOIN
    medicines AS med ON m.manufacturer_id = med.manufacturer_id
GROUP BY
    m.manufacturer_name;
```

Этот запрос соединяет таблицы manufacturers и medicines, а затем группирует результат по названию производителя.

Ключевая логика находится внутри агрегатной функции COUNT(). Для каждого лекарства CASE проверяет его количество на складе. Если оно меньше 150, первый CASE возвращает 1 (которое COUNT посчитает), а второй CASE возвращает NULL (которое COUNT проигнорирует), и наоборот.

Таким образом, мы получаем условный подсчет в двух разных категориях для каждой группы.

	A-Z manufacturer_name	123 low_stock_medicines	123 high_stock_medicines
1	ООО "Фармстандарт"	1	2
2	Bayer AG	1	1

Рисунок 2 – Результат запроса с использованием условной агрегации

2. Использование подзапросов

Подзапрос (subquery) – это **SELECT**-запрос, вложенный внутрь **другого SQL-оператора**. Он позволяет использовать результаты одного запроса для фильтрации или вычисления данных в другом.

Это мощный механизм, позволяющий строить динамические условия, которые зависят от текущего состояния данных в базе.

2.1 Скалярный подзапрос

Скалярный подзапрос возвращает одно единственное значение (одну строку, один столбец).

Его основное применение – в предложении **WHERE** для сравнения значения в столбце с неким **вычисленным показателем**, например, средним, максимальным или минимальным значением по всей таблице.

Листинг 3. Лекарства, цена которых выше средней

```
SELECT
    name,
    price
FROM
    medicines
WHERE
    price > (SELECT AVG(price) FROM medicines);
```

Сначала СУБД выполняет **внутренний подзапрос** (SELECT AVG(price) FROM medicines), который вычисляет одно-единственное число – **среднюю цену** всех лекарств.

Затем выполняется **внешний запрос**, который выбирает из таблицы medicines только те строки, у которых значение в столбце price больше, чем результат, возвращенный подзапросом.

	A-Z name ▼	123 price ▼
1	Аспирин	120
2	Витамин С	250

Рисунок 3 – Результат запроса с использованием скалярного подзапроса

2.2 Многострочный подзапрос с IN

Многострочный подзапрос возвращает **набор значений** в виде одного **столбца**, но нескольких строк (то есть список). Этот список затем используется в основном запросе для **фильтрации**.

Оператор **IN** позволяет проверить, присутствует ли значение из строки внешнего запроса в этом списке, сгенерированном подзапросом.

Листинг 4. Покупатели, которые совершали покупки за последний год

```
-- Найти имена и фамилии всех покупателей,  
-- которые совершали покупки за последний год  
SELECT  
    first_name,  
    last_name  
FROM  
    customers  
WHERE  
    customer_id IN (  
        SELECT customer_id  
        FROM sales  
        WHERE sale_date >= (CURRENT_DATE - INTERVAL '1 year')  
    );
```

Сначала выполняется **внутренний подзапрос**, использующий функции работы с датами: **CURRENT_DATE** получает текущую дату, **INTERVAL '1 year'** формирует интервал в один год.

Вычитая интервал из текущей даты, мы получаем дату **ровно год назад**. Таким образом, **подзапрос** отбирает ID всех покупателей, у которых есть продажи не ранее этой даты.

Затем **внешний запрос** по этому списку ID выбирает из таблицы customers имена и фамилии только тех покупателей, чьи customer_id присутствуют в списке, **сгенерированном подзапросом**.

	A-Z first_name	A-Z last_name
1	Иван	Иванов

Рисунок 4 – Результат запроса с использованием многострочного подзапроса с IN

2.3 Коррелированный подзапрос с EXISTS

Коррелированный (связанный) подзапрос – это подзапрос, который не может быть выполнен **независимо от внешнего запроса**, так как он ссылается на значения из его текущей строки.

Он выполняется **итеративно**, один раз для **каждой строки**, обрабатываемой **внешним запросом**.

Оператор **EXISTS** идеально подходит для таких подзапросов, так как он просто проверяет, вернул ли подзапрос хотя бы одну строку, не тратя ресурсы на чтение самих данных.

Листинг 5. Производители, выпускающие «Аспирин»

```
SELECT
    m.manufacturer_name
FROM
    manufacturers AS m
WHERE
    EXISTS (
        SELECT 1
        FROM medicines AS med
        WHERE
            med.manufacturer_id = m.manufacturer_id
        AND
            LOWER(med.name) LIKE '%аспирин%'
    );
```

Для каждого производителя *m* из внешнего запроса СУБД выполняет внутренний подзапрос.

Ключевым здесь является **условие** `med.manufacturer_id = m.manufacturer_id`, которое связывает (коррелирует) внутренний запрос с текущей строкой внешнего.

Если для производителя *m* внутренний запрос находит хотя бы одно лекарство (*med*) со словом «аспирин» в названии, **EXISTS** возвращает **TRUE**, и название этой компании включается в итоговый результат.

	A-Z manufacturer_name
1	Bayer AG

Рисунок 5 – Результат запроса с использованием коррелированного подзапроса с **EXISTS**

2.4 Альтернативное решение с JOIN

Многие задачи, решаемые с помощью подзапросов, можно также решить с использованием операторов соединения (**JOIN**).

JOIN физически «склеивает» таблицы по заданному условию, а затем уже к объединенному результату применяются фильтры.

Выбор между **JOIN** и подзапросом часто зависит от специфики задачи и читаемости кода, но важно уметь использовать оба подхода.

Листинг 6. Поиск производителей с помощью JOIN

```
-- Решим задачу из Листинга 5 с помощью JOIN
SELECT DISTINCT
    m.manufacturer_name
FROM
    manufacturers AS m
JOIN
    medicines AS med
ON
    m.manufacturer_id = med.manufacturer_id
WHERE
    LOWER(med.name) LIKE '%аспирин%';
```

Оператор **JOIN** создает временную таблицу, содержащую только те строки, где производитель связан с каким-либо лекарством.

Из этого объединенного набора отбираются только те строки, где название лекарства соответствует шаблону.

Так как один производитель может выпускать несколько лекарств, подходящих под условие, его имя может появиться в результате несколько раз. **DISTINCT** «схлопывает» все дубликаты, оставляя только уникальные названия производителей.

	A-Z manufacturer_name
1	Bayer AG

Рисунок 6 – Результат запроса с использованием альтернативного решения с JOIN

3. Использование обобщенных табличных выражений (CTE)

CTE (Common Table Expressions), вводимые с помощью ключевого слова **WITH**, позволяют определить временный, именованный результирующий набор, на который можно ссылаться в последующих частях запроса.

Они **не сохраняются в базе данных** и существуют только на время выполнения одного запроса.

CTE значительно улучшают **читаемость** и структурированность сложных запросов.

3.1 Стандартные CTE

Стандартные CTE служат для декомпозиции сложного запроса на простые, последовательные логические шаги.

Вместо того чтобы вкладывать один подзапрос в другой, создавая трудночитаемую "матрешку", мы можем определить каждый шаг как отдельный CTE, дать ему понятное имя и затем соединять их в финальном запросе.

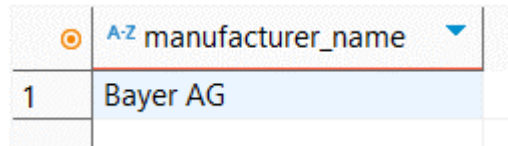
Листинг 7. Поиск производителей «Аспирина» через CTE

```
WITH aspirin_producers AS (  
    SELECT DISTINCT manufacturer_id  
    FROM medicines  
    WHERE LOWER(name) LIKE '%аспирин%'  
)  
SELECT  
    m.manufacturer_name  
FROM  
    manufacturers AS m  
JOIN  
    aspirin_producers AS ap ON m.manufacturer_id = ap.manufacturer_id;
```

Этот запрос решает ту же задачу, что и в пункте 2.3.

Сначала выполняется блок **WITH**, который создает временный именованный набор `aspirin_producers`, содержащий ID всех производителей аспирина.

Затем выполняется основной, более простой и понятный запрос, который работает с `aspirin_producers` как с обычной таблицей: он соединяет (**JOIN**) ее с таблицей `manufacturers`, чтобы получить названия компаний по их **ID**.



	A-Z manufacturer_name
1	Bayer AG

Рисунок 7 – Результат запроса

3.2 Рекурсивные CTE

Рекурсивный **CTE (WITH RECURSIVE)** – это специальный вид **CTE**, который может ссылаться сам на себя. Он является единственным стандартным способом в SQL для обхода иерархических или древовидных структур данных, таких как организационная структура компании, файловая система или дерево категорий товаров.

В демонстрационной базе данных таблица `pharmacists` содержит такую структуру через поле `manager_id`, которое ссылается на `pharmacist_id` в этой же таблице.

Листинг 8. Вывод всей иерархии подчиненности фармацевтов

```
WITH RECURSIVE subordinates AS (  
  
    -- 1. Якорный (начальный) запрос: находим "корень" иерархии  
    SELECT  
        pharmacist_id,  
        first_name,  
        last_name,  
        manager_id,  
        -- Устанавливаем начальный уровень иерархии  
        0 AS level  
    FROM  
        pharmacists  
    WHERE  
        -- Начинаем с самого верхнего руководителя  
        manager_id IS NULL  
  
    UNION ALL
```

```

-- 2. Рекурсивный запрос: присоединяем прямых подчиненных
SELECT
    p.pharmacist_id,
    p.first_name,
    p.last_name,
    p.manager_id,
    -- Увеличиваем уровень иерархии на 1
    s.level + 1 AS level
FROM
    pharmacists AS p
JOIN
    subordinates AS s ON p.manager_id = s.pharmacist_id
)
SELECT
    -- Используем LPAD для наглядного отображения иерархии с отступами
    LPAD(' ', level * 4, ' ') || first_name || ' ' || last_name AS
    hierarchy,
    level
FROM subordinates;

```

Работа этого запроса делится на две логические части, объединенные **UNION ALL**, и выполняется итеративно:

1. Якорный запрос (Anchor Member)

Эта часть выполняется **один раз** в самом начале.

Она находит "корневые" элементы иерархии (в нашем случае – фармацевта, у которого manager_id IS NULL, то есть главного руководителя) и формирует первоначальный набор результатов.

Мы также **добавляем столбец level** со значением **0**, чтобы обозначить верхний уровень иерархии.

2. Рекурсивный запрос (Recursive Member)

Эта часть выполняется **итеративно**.

На **первой итерации** она берёт результаты якорного запроса (обозначенные как s) и присоединяет (JOIN) к ним тех фармацевтов (p), у которых manager_id совпадает с pharmacist_id из s.

Иными словами, она находит прямых подчиненных для результатов предыдущего шага.

При этом **уровень (level) увеличивается** на единицу (s.level + 1).

На следующей итерации процесс повторяется: запрос берет результаты предыдущей рекурсивной итерации (сотрудников с *level* = 1) и ищет уже их прямых подчиненных, присваивая им *level* = 2, и так далее.

Роль UNION ALL заключается в простом добавлении результатов каждой новой итерации рекурсивного запроса к общему набору результатов.

Использование UNION ALL (а не UNION) критически важно, так как он не тратит ресурсы на проверку уникальности строк, что значительно ускоряет выполнение.

Условие завершения: рекурсия прекращается автоматически, когда рекурсивный запрос перестает возвращать новые строки. Это происходит, когда JOIN больше не может найти совпадений, то есть, когда он доходит до самого нижнего уровня иерархии – сотрудников, у которых нет подчиненных.

	A-Z hierarchy	level
1	Иван Сорокин	0
2	Алексей Петров	1
3	Марина Иванова	1
4	Евгений Смирнов	2
5	Александр Орлов	2

Рисунок 8 – Результат запроса с рекурсивным CTE

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Объясните разницу между фильтрацией данных с помощью предложения **WHERE** и техникой условной агрегации с использованием **CASE** внутри агрегатной функции (например, **COUNT** или **SUM**). Какую задачу решает каждый из подходов?
2. В чем заключается фундаментальное отличие между коррелированным и некоррелированным подзапросом с точки зрения логики их выполнения СУБД и потенциального влияния на производительность?
3. Приведите пример бизнес-задачи для вашей предметной области, для решения которой использование оператора **EXISTS** будет более семантически верным и потенциально более эффективным, чем **IN**.
4. Опишите общую структуру рекурсивного **СТЕ**. Какие две части его составляют, каково их назначение и как они взаимодействуют? Как СУБД определяет, когда необходимо остановить рекурсивный процесс?
5. Сравните использование подзапроса в предложении **FROM** и обобщенного табличного выражения (**СТЕ**). В какой ситуации использование **СТЕ** является предпочтительным и почему?

КРАТКИЙ СПРАВОЧНЫЙ МАТЕРИАЛ

1. Функции SQL

1.1 Функции для работы с числами

Таблица 1. Функции для работы с числами

Функция	Описание
POW (num, power)	Возведение числа в степень.
SELECT POW(3, 2); Результат: 9	
SQRT (num)	Вычисление квадратного корня.
SELECT SQRT(16); Результат: 16	
CEIL (num)	Округление числа до ближайшего целого в большую сторону.
SELECT CEIL(75.1); Результат: 76	
FLOOR (num)	Округление числа до ближайшего целого в меньшую сторону.
SELECT FLOOR(75.1); Результат: 75	
ROUND (num, [decimals])	Математическое округление до указанного знака.
SELECT ROUND (75.56, 1); Результат: 75.6	
CAST (value AS type)	Явное преобразование одного типа данных в другой.
SELECT CAST('123.4' AS NUMERIC); Результат: 123.4	

1.2 Функции для работы с датой и временем

Таблица 2. Функции для работы с датой и временем

Функция	Описание
CURRENT_DATE	Возвращает текущую дату.
SELECT CURRENT_DATE; Результат: 2025-09-16	
CURRENT_TIME	Возвращает текущее время с часовым поясом.
SELECT CURRENT_TIME; Результат: 14:45:31.847 +0300	
NOW() / CURRENT_TIMESTAMP	Возвращает текущие дату и время с часовым поясом.
SELECT CURRENT_TIMESTAMP - INTERVAL '1 month'; Результат: 2025-08-16 14:45:31.847 +0300	
AGE (date1, [date2])	Вычисляет разницу между двумя датами (или между date1 и NOW()).
SELECT AGE('2025-01-01', '2024-11-20'); Результат: 1 mon 11 days	
EXTRACT (part FROM date)	Извлекает часть из даты (например, YEAR, MONTH, DAY).
SELECT EXTRACT(YEAR FROM '2025-09-21'::DATE); Результат: 2025	
TO_DATE (str, format)	Преобразует строку в дату в соответствии с форматом.
SELECT TO_DATE('21 09 2025', 'DD MM YYYY'); Результат: 2025-09-16	

1.3 Функции для работы со строками

Таблица 3. Функции для работы со строками

Функция	Описание
LENGTH (str)	Возвращает длину строки.
SELECT LENGTH('Аспирин'); Результат: 7	
LOWER (str) / UPPER (str)	Преобразует строку к нижнему / верхнему регистру.
SELECT LOWER('Аспирин'); Результат: 'аспирин' SELECT UPPER('Аспирин'); Результат: 'АСПИРИН'	
TRIM (str)	Удаляет пробелы в начале и в конце строки.
SELECT TRIM (' Аспирин '); Результат: 'Аспирин'	
SUBSTRING (str FROM pos FOR len)	Извлекает подстроку.
SELECT SUBSTRING('Парацетамол' FROM 1 FOR 5); Результат: 'Парац'	
REPLACE (str, from, to)	Заменяет все вхождения подстроки.
SELECT REPLACE('Аспирин', 'ин', 'ИН'); Результат: 'АспирИН'	
CONCAT (str1, str2, ...)	Объединяет строки.
SELECT CONCAT ('Асп', 'Ирин'); Результат: 'АспИрин'	

2. Условная логика и подзапросы

2.1 Оператор CASE

Оператор **CASE** – это аналог конструкции **if-then-else** в SQL, позволяющий реализовать условную логику непосредственно внутри запроса.

Основные сценарии использования:

- **Категоризация данных:** присвоение записям меток на основе их атрибутов (например, 'дорогой'/'дешевый').

Листинг 9. Пример категоризации данных

```
SELECT
  CASE
    WHEN price > 100 THEN 'Дорого'
    ELSE 'Доступно'
  END AS category
FROM medicines;
```

- **Условная агрегация:** вычисление агрегатных функций (SUM, COUNT) только для строк, удовлетворяющих условию, путем помещения CASE внутрь агрегатной функции.

Листинг 10. Пример условной агрегации

```
-- Подсчет количества лекарств с истекающим сроком годности.

SELECT
  COUNT(CASE WHEN expiration_date < (CURRENT_DATE + INTERVAL '1
month') THEN 1 END)
FROM medicines;
```

2.2 Подзапросы (Subqueries)

Подзапрос – это **SELECT**-запрос, вложенный в **другой** SQL-оператор. Позволяет использовать результаты одного запроса для формирования условий или данных в другом.

Таблица 4. Сравнение типов подзапросов

Тип подзапроса	Что возвращает	Основное применение	Пример оператора
Скалярный	Одно значение (1 строка, 1 столбец)	Сравнение со значением (=, >, <) в WHERE или HAVING .	WHERE price > (SELECT AVG (price) FROM ...)
Многострочный	Список значений (N строк, 1 столбец)	Проверка вхождения значения в список.	WHERE id IN (SELECT id FROM ...)
Табличный	Таблица (N строк, N столбцов)	Как источник данных для FROM или JOIN .	FROM (SELECT ...) AS subquery
Коррелированный	Результат зависит от внешнего запроса	Проверка существования связанной записи для каждой строки внешнего запроса.	WHERE EXISTS (SELECT 1 FROM ...)

2.3 Сравнение EXISTS Vs INNER JOIN

Основное правило: если нужно ответить на вопрос «да/нет» (*существует ли связь?*), используйте **EXISTS**. Если вам нужны данные из связанной таблицы, используйте **JOIN**.

Таблица 5. Сравнение EXISTS Vs INNER JOIN

Критерий	WHERE EXISTS (подзапрос)	INNER JOIN
Основная цель	Проверить факт существования хотя бы одной связанной записи.	Получить и использовать данные из связанной таблицы.
Логика выполнения	Полусоединение (Semi-join). Для каждой строки внешней таблицы подзапрос ищет первое совпадение.	Полное соединение. Находит все совпадающие строки и добавляет их в промежуточный результат.

	Как только оно найдено, поиск прекращается, и возвращается TRUE.	
Возвращаемые данные	Не возвращает данные из подзапроса. Только TRUE или FALSE.	Возвращает столбцы из обеих таблиц.
Влияние на результат	Не дублирует строки из внешней таблицы.	Может дублировать строки, если в правой таблице нашлось несколько совпадений для одной строки из левой. Часто требует DISTINCT.
Когда использовать	Идеально для: «Найти всех производителей, у которых есть хотя бы одно лекарство». Нам не важно, сколько и каких, важен сам факт наличия.	Идеально для: «Вывести всех производителей и названия их лекарств». Нам нужны данные из второй таблицы.

3. Обобщенные табличные выражения (CTE)

CTE (Common Table Expressions) – это, по сути, временная именованная таблица, которая существует только на время выполнения одного SQL-запроса. Она определяется с помощью ключевого слова **WITH** в самом начале запроса.

3.1 Стандартный CTE

Стандартный CTE используется для **декомпозиции** – разбиения одного большого и сложного запроса на несколько последовательных, логически понятных шагов.

Это делает код значительно более читаемым и простым в отладке по сравнению с использованием множества вложенных подзапросов.

Ключевые преимущества:

- **Читаемость:** вместо "матрешки" из вложенных SELECT, вы получаете последовательность блоков: "сначала мы вычислили А, затем на основе А вычислили Б, и в конце получили результат из Б".
- **Многократное использование:** на один и тот же CTE можно ссылаться несколько раз в последующих частях запроса, что избавляет от необходимости копировать один и тот же код.

Листинг 11. Пример

```
WITH HighStockMedicines AS (  
    SELECT name, price FROM medicines WHERE quantity_in_stock > 100  
)  
SELECT * FROM HighStockMedicines WHERE price < 50;
```

3.2 Рекурсивный CTE

Это особый вид CTE, который может **ссылаться сам на себя**, что позволяет ему итеративно строить результирующий набор данных. Это стандартный и самый эффективный способ в SQL для работы с иерархическими (начальник-подчиненный) или древовидными структурами (категории-подкатегории).

Структура и принцип работы:

- **Якорный запрос (база рекурсии):** это SELECT, который выполняется всего один раз. Он находит «корневые» элементы иерархии (*например, генерального директора, у которого нет начальника*) и формирует стартовый набор строк.
- **UNION ALL:** оператор, который «склеивает» результаты. **ВАЖНО:** почти всегда используется именно UNION ALL, так как он значительно быстрее (*не проверяет на дубликаты, что в рекурсии обычно не требуется*).

- **Рекурсивный запрос (*шаг рекурсии*):** это **SELECT**, который ссылается на имя **самого СТЕ**. Он выполняется **многократно**. На каждой итерации он берет строки, полученные на предыдущем шаге, и присоединяет к ним следующий уровень иерархии (*например, находит прямых подчиненных для менеджеров, найденных на прошлом шаге*).

Листинг 12. Пример – генерация чисел от 1 до 5.

```
WITH RECURSIVE numbers(n) AS (  
    -- Якорь: начинаем с 1  
    SELECT 1  
    UNION ALL  
    -- Рекурсия: прибавляем 1, пока n < 5 (условие остановки)  
    SELECT n + 1 FROM numbers WHERE n < 5  
)  
SELECT * FROM numbers;
```

Главная проблема – бесконечная рекурсия.

Если в рекурсивном запросе нет условия остановки, он будет выполняться бесконечно, пока не исчерпает ресурсы сервера.

Решение: наличие **чёткого условия завершения**.

В рекурсивном запросе всегда должно быть условие, которое в какой-то момент перестанет выполняться. Например, рекурсия прекратится сама собой, когда дойдет до самого нижнего уровня иерархии (сотрудников, у которых нет подчиненных). В качестве дополнительной защиты рекомендуется добавлять счетчик уровня вложенности и ограничивать его (**WHERE level < 100**).