

# Лекция 8.

## **Язык SQL - продолжение**

**Обобщенные табличные  
выражения  
(Common Table Expression, CTE)**

## Обобщенные табличные выражения

CTE - именованный подзапрос, указываемый в предложении WITH в верхней части запроса SELECT, INSERT, UPDATE или DELETE.

Действует аналогично временному представлению (view).

## 1. «Табличное выражение» (Table Expression)

В реляционной теории и SQL **табличное выражение** — это любая конструкция, которая **возвращает результат в виде таблицы** (то есть набора строк и колонок), даже если физически таблицы не существует.

К табличным выражениям относятся:

- Таблицы в базе данных ( `SELECT * FROM employees` );
- Представления ( `VIEW` );
- Подзапросы в `FROM` :

```
sql
1 SELECT * FROM (SELECT id, name FROM users WHERE active = 1) AS active_users;
```

- И, конечно, **CTE**.

То есть «**табличное выражение**» = «**что-то, что ведёт себя как таблица в рамках запроса**».

💡 Важно: это **логическая таблица**, а не обязательно физическое хранилище. Она существует только на время выполнения запроса. "

## 2. «Общее» (Common)

Слово «**common**» означает, что это выражение:

- **Определяется один раз** в начале запроса (в блоке `WITH`);
- **Может использоваться многократно** в основном запросе (в отличие от подзапроса, который пришлось бы дублировать);
- **Доступно «общему» использованию** внутри одного SQL-запроса — отсюда и «common».

Пример:

```
sql
1 WITH sales_summary AS (
2     SELECT region, SUM(amount) AS total
3     FROM sales
4     GROUP BY region
5 )
6 SELECT * FROM sales_summary WHERE total > 1000
7 UNION ALL
8 SELECT * FROM sales_summary WHERE region = 'West';
```

Здесь `sales_summary` — **общее** табличное выражение: оно определено один раз, но использовано дважды.

## Итог

- «**Табличное выражение**» — потому что результат выглядит и ведёт себя как таблица (набор строк и колонок).
- «**Общее**» (**Common**) — потому что оно определяется один раз и может использоваться многократно **внутри одного запроса**.
- CTE — это **именованный, локальный, нематериализованный подзапрос**, улучшающий читаемость и структуру SQL.

Таким образом, название **точно отражает суть**: это **общее (в рамках запроса) выражение, возвращающее таблицу**.

### 1. Нематериализованный подзапрос (inline / folded)

- СУБД не вычисляет подзапрос отдельно.
- Вместо этого оптимизатор **раскрывает («встраивает»)** его содержимое в основной запрос и оптимизирует всё целиком.

# Обычные и рекурсивные СТЕ

## **Обычные СТЕ:**

- Действуют как представления (view) с учетом того, что к СТЕ можно обращаться только в том запросе, где они объявлены.
- Заменяют подзапросы, позволяя упростить структуру запроса.

## **Рекурсивные СТЕ (обращаются сами к себе):**

- Генерирование последовательности значений.
- Выполнение сложных преобразования данных.
- Обработка иерархических данных.

# Обычные СТЕ

# Общая структура CTE

```
sql
1 WITH имя_cte AS (
2     SELECT ...
3     FROM ...
4     WHERE ...
5 )
6 SELECT ...
7 FROM имя_cte
8 WHERE ...;
```

- `WITH` — ключевое слово, начинающее определение CTE.
- `имя_cte` — произвольное имя, по которому можно обращаться к результату подзапроса.
- Внутри скобок — обычный `SELECT` (без `ORDER BY`, если только не используется `LIMIT` или `TOP` в некоторых СУБД).
- После определения CTE можно использовать его в основном запросе как обычную таблицу или представление.

## Общая структура CTE

Можно определить несколько CTE через запятую:

sql

```
1 WITH cte1 AS (...),  
2     cte2 AS (...)  
3 SELECT ...  
4 FROM cte1  
5 JOIN cte2 ...
```

# Сходство с функциями в языках программирования

1. **Инкапсулируют логику** — как функция скрывает детали реализации, так и CTE скрывает сложный подзапрос за понятным именем.
2. **Повышают читаемость** — вместо вложенных подзапросов вы пишете последовательные, именованные блоки.
3. **Локальная область видимости** — CTE существует только в рамках одного запроса, как локальная переменная или вспомогательная функция внутри другой функции.
4. **Могут использоваться многократно** — если CTE определён один раз, его можно использовать несколько раз в основном запросе (в отличие от подзапроса в `FROM`, который пришлось бы дублировать).

Однако, в отличие от функций, CTE не компилируются отдельно, не кэшируют результат (в большинстве СУБД они "встраиваются" в основной запрос), и не поддерживают параметризацию.

# СТЕ для Замены подзапросов

**Задача:** Найти клиентов, у которых общая сумма заказов выше средней суммы заказов по всем клиентам.

× С подзапросами (вложено):

```
sql
1 SELECT customer_id, total_spent
2 FROM (
3     SELECT customer_id, SUM(amount) AS total_spent
4     FROM orders
5     GROUP BY customer_id
6 ) AS customer_totals
7 WHERE total_spent > (
8     SELECT AVG(total_spent)
9     FROM (
10        SELECT customer_id, SUM(amount) AS total_spent
11        FROM orders
12        GROUP BY customer_id
13    ) AS inner_totals
14 );
```

→ Подзапрос дублируется! Трудно читать и поддерживать.

## ✓ C CTE:

```
sql
1 ✓ WITH customer_totals AS (
2     SELECT customer_id, SUM(amount) AS total_spent
3     FROM orders
4     GROUP BY customer_id
5 ),
6 avg_total AS (
7     SELECT AVG(total_spent) AS avg_spent
8     FROM customer_totals
9 )
10 SELECT ct.customer_id, ct.total_spent
11 FROM customer_totals ct
12 CROSS JOIN avg_total at
13 WHERE ct.total_spent > at.avg_spent;
```

→ Каждый шаг ясен, дублирования нет, легко изменить логику.

## Вывод

СТЕ превращают «вложенную логику» в «линейную последовательность шагов», что:

- упрощает чтение,
- облегчает отладку,
- снижает вероятность ошибок,
- делает SQL похожим на декларативный рассказ: «Сначала сделай это, потом — то, и в итоге получишь результат».

Это особенно ценно в аналитических и бизнес-запросах, где логика часто состоит из нескольких этапов агрегации и фильтрации.

“ ✨ **Правило большого пальца:** если подзапрос повторяется или мешает понять логику — вынеси его в СТЕ. ”

# Рекурсивные СТЕ

## Зачем нужны рекурсивные CTE?

Они позволяют работать с **данными, имеющими древовидную или иерархическую структуру**, например:

- Организационная структура (сотрудник → руководитель → руководитель руководителя...)
- Категории товаров (электроника → ноутбуки → игровые ноутбуки)
- Маршруты или цепочки зависимостей
- Генерация последовательностей (дат, чисел)

**Без рекурсии** такие задачи пришлось бы решать с помощью:

- циклов в приложении,
- заранее известной глубины вложенностей,
- сложных JOIN'ов (если уровень вложенности фиксирован).

Рекурсивные CTE делают это **элегантно и универсально**.

Поскольку SQL не поддерживает циклы напрямую, рекурсивные CTE дают **декларативный способ** обходить иерархии и выполнять итерации **без использования процедурного кода** (например, PL/pgSQL, T-SQL и т.п.).

```
WITH [RECURSIVE] cte-table-name AS (  
  initial-select  
  UNION [ALL]  
  recursive-select)  
cte-select
```

Рекурсивный CTE состоит из двух частей:

1. **Якорь (anchor member)** — начальное нерекурсивное выражение, задающее стартовый набор строк.
2. **Рекурсивная часть (recursive member)** — выражение, которое ссылается на сам CTE и добавляет новые строки на основе предыдущих.

Эти части объединяются с помощью `UNION ALL`.

```
WITH [RECURSIVE] cte-table-name AS (  
  initial-select  
  UNION [ALL]  
  recursive-select)  
cte-select
```

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<10)  
SELECT * FROM ten;
```

**Это точно рекурсия??**

x
—
1
2
3
5
6
7
8
9
10

# Рекурсивные функции

Рекурсия — это когда функция вызывает саму себя для решения задачи.

Чтобы рекурсия работала корректно, в ней обязательно должны быть:

1. **Базовый случай** — условие, при котором функция перестаёт вызывать себя и возвращает результат напрямую.
2. **Рекурсивный случай** — часть, где функция вызывает саму себя с изменёнными (обычно уменьшенными) аргументами, приближаясь к базовому случаю.

```
python
```

```
1 def factorial(n):  
2     # Базовый случай  
3     if n == 0:  
4         return 1  
5     # Рекурсивный случай  
6     return n * factorial(n - 1)  
7  
8 # Пример использования  
9 print(factorial(5)) # Выведет: 120
```

# Рекурсивные функции

python

```
1 def factorial(n):
2     # Базовый случай
3     if n == 0:
4         return 1
5     # Рекурсивный случай
6     return n * factorial(n - 1)
7
8 # Пример использования
9 print(factorial(5)) # Выведет: 120
```

Как это работает шаг за шагом (для `factorial(3)`):

1. `factorial(3)` → вызывает `3 * factorial(2)`
2. `factorial(2)` → вызывает `2 * factorial(1)`
3. `factorial(1)` → вызывает `1 * factorial(0)`
4. `factorial(0)` → возвращает `1` (базовый случай)
5. Теперь всё "раскручивается":

- `factorial(1) = 1 * 1 = 1`
- `factorial(2) = 2 * 1 = 2`
- `factorial(3) = 3 * 2 = 6`

# Рекурсивные функции

python

```
1 def factorial(n):
2     # Базовый случай
3     if n == 0:
4         return 1
5     # Рекурсивный случай
6     return n * factorial(n - 1)
7
8 # Пример использования
9 print(factorial(5)) # Выведет: 120
```

Как это работает шаг за шагом (для `factorial(3)`):

1. `factorial(3)` → вызывает `3 * factorial(2)`
2. `factorial(2)` → вызывает `2 * factorial(1)`
3. `factorial(1)` → вызывает `1 * factorial(0)`
4. `factorial(0)` → возвращает `1` (базовый случай)
5. Теперь всё "раскручивается":

- `factorial(1) = 1 * 1 = 1`
- `factorial(2) = 2 * 1 = 2`
- `factorial(3) = 3 * 2 = 6`

```
WITH RECURSIVE ten(x) AS (
    SELECT 1
    UNION ALL
    SELECT x+1 FROM ten WHERE x<10)
SELECT * FROM ten;
```

x  
—  
1  
2  
3  
5  
6  
7  
8  
9  
10

**Как работает эта рекурсия?**

## 🧠 Почему это называют «рекурсией», если это итерация?

Термин «рекурсивный CTE» — условное название, принятое в стандарте SQL. Оно отражает **логическую структуру запроса**: CTE ссылается на само себя в своём определении.

Но **физически** это:

- **Фиксированный цикл** (обычно реализованный как `while` в коде СУБД),
- Работающий с **реляционными операциями** (JOIN, SELECT и т.д.),
- Без вложенных вызовов.

Поэтому точнее говорить:

“Рекурсивный CTE — это синтаксический сахар для итеративного процесса обхода данных, выраженный в декларативной форме.”

# Настоящая рекурсия и СТЕ

Характеристика	Настоящая рекурсия в ЯП	Рекурсивный СТЕ (SQL)
Модель вычислений	Вызов функции – стек вызовов	Итеративное расширение множества
Хранение состояния	В стеке (локальные переменные)	В промежуточной временной таблице
Условие завершения	Базовый случай в коде	Пустой результат рекурсивной части
Возможность «возврата»	Да (возврат значения вверх по стеку)	Нет – только накопление строк
Поддержка произвольной логики	Да (if, циклы, исключения и т.д.)	Нет – только реляционные операции
Возможность зацикливания	Бесконечная рекурсия – переполнение стека	Бесконечная рекурсия – лимит глубины

```
WITH [RECURSIVE] cte-table-name AS (  
  initial-select  
  UNION [ALL]  
  recursive-select)  
cte-select
```

## Алгоритм работы рекурсивного CTE в SQLite

1. Выполняется начальная выборка (initial-select), результат помещается в очередь и в результирующую таблицу.
2. Пока в очереди есть элементы, из очереди извлекается одна строка S и добавляется в рекурсивную таблицу.
3. Выполняется рекурсивная выборка (recursive-select), при этом считается, что рекурсивная таблица состоит из одной строки S. Результат выборки добавляется в очередь и в результирующую таблицу.
4. Переходим к шагу 2.

LIMIT в рекурсивном запросе определяет максимальное число строк, которые добавляются в рекурсивную таблицу на шаге 2.

# Рекурсивные CTE

1

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
SELECT * FROM ten;
```

Очередь

x
—
<b>1</b>

Рекурсивная ten

x
—

Результат ten

x
—
<b>1</b>

# Рекурсивные CTE

2

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
SELECT * FROM ten;
```

Очередь

x
—

Рекурсивная ten

x
—
<b>1</b>

Результат **ten**

x
—
1

# Рекурсивные CTE

3

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
SELECT * FROM ten;
```

Очередь

x
—
2

Рекурсивная ten

x
—
1

Результат ten

x
—
1
2

# Рекурсивные CTE

2

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
SELECT * FROM ten;
```

Очередь

x
—

Рекурсивная ten

x
—
2

Результат **ten**

x
—
1
2

# Рекурсивные CTE

3

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
  SELECT * FROM ten;
```

Очередь

x
—
<b>3</b>

Рекурсивная ten

x
—
2

Результат **ten**

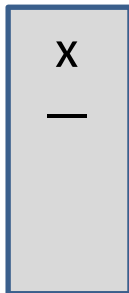
x
—
1
2
<b>3</b>

# Рекурсивные CTE

2

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
SELECT * FROM ten;
```

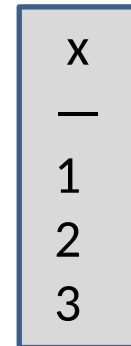
Очередь



Рекурсивная ten



Результат ten



# Рекурсивные CTE

3

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
SELECT * FROM ten;
```

Очередь

x
—
4

Рекурсивная ten

x
—
3

Результат ten

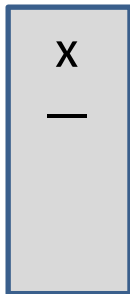
x
—
1
2
3
4

# Рекурсивные CTE

2

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
SELECT * FROM ten;
```

Очередь



Рекурсивная ten



Результат ten



# Рекурсивные CTE

3

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
  SELECT * FROM ten;
```

Очередь

x
—
<b>5</b>

Рекурсивная ten

x
—
4

Результат ten

x
—
1
2
3
4
<b>5</b>

# Рекурсивные CTE

2

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
SELECT * FROM ten;
```

Очередь

x
—

Рекурсивная ten

x
—
5

Результат ten

x
—
1
2
3
4
5

# Рекурсивные CTE

3

```
WITH RECURSIVE ten(x) AS (  
  SELECT 1  
  UNION ALL  
  SELECT x+1 FROM ten WHERE x<5)  
  SELECT * FROM ten;
```

Очередь

x
—

Рекурсивная ten

x
—
5

Результат ten

x
—
1
2
3
4
5

В большинстве других СУБД рекурсивные CTE реализованы иначе (но конечный результат будет тем же)

1. Выполняется `anchor member` (начальный запрос).

→ Его результат:

- Добавляется в итоговую таблицу (`T`).
- Становится первым "слоем" (`W` — working set).

2. Повторять, пока `W` не пуст:

- Выполнить `recursive member`, где ссылка на CTE в `FROM` означает весь текущий `W` (не одну строку!).
- Получить новый результат `R`.
- Добавить все строки из `R` в итоговую таблицу `T`.
- Сделать `W=R` (новый рабочий набор — результат текущей итерации).

3. Вернуть `T`.

“ Это итеративный процесс по уровням (слоям), а не по отдельным строкам.”

# Рекурсивные СТЕ

Пусть  $T$  — итоговый результат,  $W$  — рабочий набор.

1.  $W = \text{anchor\_query}()$

2.  $T = W$

3. Пока  $W \neq \emptyset$  :

- $W = \text{recursive\_query}(W)$  ← здесь  $W$  — таблица, а не строка

- $T = T \cup W$

4. Вернуть  $T$

# Обработка иерархических структур

---

## Отдел 1

Сотрудник 1

Сотрудник 2

...

## Сектор 1

Сотрудник 1

Сотрудник 2

...

...

## Отдел 2

Сотрудник 1

...

## Сектор 1

Сотрудник 1

Сотрудник 2

...

...

Необходимо выделить всех сотрудников определенного отдела (вне зависимости от того, уровня вложенности)

# Обработка иерархических структур

```
1 HR (id=1, department)
2 |— Diana (id=104)
3 |— Recruiting (id=6, sector)
4 |   |— Eve (id=105)
5
6 IT (id=2, department)
7 |— Alice (id=101)
8 |— Backend (id=3, sector)
9 |   |— Bob (id=102)
10 |       |— DevOps (id=5, sector)
11 |           |— Charlie (id=103)
12 |— Frontend (id=4, sector)
```

## 💡 Примечание по структуре: "

- Сотрудники, у которых `unit_id` ссылается на **отдел напрямую** (например, Alice в IT, Diana в HR), показаны **на том же уровне, что и сам отдел**.
- Сотрудники, прикрепленные к **секторам**, показаны **внутри соответствующего сектора**.
- Вложенность секторов отражена через отступы (DevOps — внутри Backend).

# Обработка иерархических структур

## 1. Структура таблиц

Для моделирования такой структуры удобно использовать **единую иерархическую таблицу для всех узлов** (и отделов, и секторов), а сотрудников — отдельно.

Таблица `units` — все организационные единицы (и отделы, и сектора)

ID	NAME	PARENT_ID	TYPE
1	HR	NULL	'department'
2	IT	NULL	'department'
3	Backend	2	'sector'
4	Frontend	2	'sector'
5	DevOps	3	'sector'
6	Recruiting	1	'sector'

- `type` помогает отличать отделы от секторов (опционально, но полезно).
- Корневые узлы ( `parent_id IS NULL` ) — это отделы верхнего уровня.

# Обработка иерархических структур

Таблица `employees`

ID	NAME	UNIT_ID
101	Alice	2
102	Bob	3
103	Charlie	5
104	Diana	1
105	Eve	6

“Поле `unit_id` ссылается на `units.id` .”

# Обработка иерархических структур

---

## Задача

Найти **всех сотрудников**, которые относятся к отделу IT (id = 2), включая:

- тех, кто прикреплен напрямую к IT,
  - тех, кто в Backend, Frontend,
  - тех, кто в DevOps (вложен в Backend).
-

# Обработка иерархических структур

## Решение: рекурсивный CTE

sql

```
1 WITH RECURSIVE it_subtree AS (  
2     -- Якорь: сам отдел IT  
3     SELECT id  
4     FROM units  
5     WHERE id = 2 -- или WHERE name = 'IT' AND parent_id IS NULL  
6  
7     UNION ALL  
8  
9     -- Рекурсия: все дочерние сектора любого уровня  
10    SELECT u.id  
11    FROM units u  
12    INNER JOIN it_subtree it ON u.parent_id = it.id  
13 )  
14 SELECT e.id, e.name, u.name AS unit_name  
15 FROM employees e  
16 JOIN it_subtree t ON e.unit_id = t.id  
17 JOIN units u ON u.id = e.unit_id;
```

## Объяснение CTE

### Часть 1: Anchor member (якорь)

sql

```
1 SELECT id
2 FROM units
3 WHERE id = 2
```

- Начинаем с корневого узла — отдела IT (id = 2).
- Это наша «точка входа» в иерархию.

## Часть 2: Recursive member (рекурсивная часть)

```
sql
1 SELECT u.id
2 FROM units u
3 INNER JOIN it_subtree it ON u.parent_id = it.id
```

- На каждой итерации ищем **все узлы**, у которых `parent_id` совпадает с `id` из уже найденных узлов.
- То есть: сначала найдём `Backend` и `Frontend` (их `parent_id = 2`), затем — `DevOps` (`parent_id = 3`), и так далее.
- Рекурсия остановится, когда не останется дочерних узлов.

Результат CTE `it_subtree` :

ID
2
3
4
5

## Часть 3: Основной запрос

```
sql
1 SELECT ...
2 FROM employees e
3 JOIN it_subtree t ON e.unit_id = t.id
4 ...
```

- Просто выбираем всех сотрудников, чей `unit_id` есть в поддереве IT.

## ✓ Результат запроса

ID	NAME	UNIT_NAME
101	Alice	IT
102	Bob	Backend
103	Charlie	DevOps

(Предполагаем, что в Frontend нет сотрудников — иначе они тоже попали бы.)

## Преимущества такого подхода

- **Гибкость:** структура поддерживает неограниченную вложенность секторов.
- **Единая модель:** отделы и сектора — один тип сущности ( `units` ), что упрощает рекурсию.
- **Производительность:** один рекурсивный обход вместо множества вложенных подзапросов.
- **Читаемость:** логика явно разделена: сначала строим поддерево, потом ищем сотрудников.

**Рекурсивный СТЕ - это не  
«настоящая» рекурсия**

## Что такое «настоящая» рекурсия?

В языках программирования (например, Python, JavaScript, Haskell):

- **Рекурсия** — это **вызов функции из самой себя**.
- Каждый вызов создаёт **новый фрейм стека**, где хранятся локальные переменные, параметры и точка возврата.
- Вычисления происходят **последовательно**: сначала «погружение» (до базового случая), потом «всплытие» (возврат результатов вверх по стеку).
- Можно передавать **произвольные состояния**, возвращать **сложные структуры**, использовать **условную логику, замыкания** и т.д.

Пример (Python):

```
python 📄 ⬇  
1 def factorial(n):  
2     if n <= 1:  
3         return 1  
4     return n * factorial(n - 1)
```

Здесь каждый вызов `factorial` — независимый, с собственным контекстом.

## “Рекурсия” в SQL (CTE)

SQL — декларативный язык, ориентированный на множества (sets) и реляционную алгебру, а не на последовательное выполнение инструкций.

Рекурсивный CTE не вызывает сам себя как функцию. Вместо этого:

“SQL-движок выполняет его итеративно, шаг за шагом, накапливая результат в промежуточной таблице.”

# SQL - резюме

# Рассмотренные возможности SQL

---

- Определение схемы данных (DDL)
- Управление данными (DML)
- Выборка и анализ данных (DQL)

# Определение схемы данных

- Создание базы данных
- Создание таблиц
- Связь таблиц друг с другом (Foreign Key – Primary Key) с помощью ограничений
- Создание индексов - не рассматривали

# Управление данными

- Вставка данных
- Изменение данных
- Удаление данных

# Выборка данных

- Выборка данных из наборов записей по разным критериям
  - Подзапросы
  - Обычные СТЕ
  - Представления
- Работа с иерархическими данными
  - Рекурсивные СТЕ

## Анализ данных

- Обработка выбранных данных
  - Группировка с агрегацией
  - Сортировка
  - Аналитические функции
- Работа с иерархическими данными
  - Рекурсивные CTE

## Не рассмотрели

- Создание индексов
- Управление транзакциями
- Управление пользователями и правами доступа
- Императивное программирование на SQL-сервере
  - триггеры
  - хранимые процедуры на сервере

# **SQL для анализа наборов данных**

# Кому нужно обрабатывать и анализировать наборы данных?

---

## 1. Аналитики (BI, продуктовые, маркетинговые)

**Задача:** понять поведение пользователей или эффективность кампаний.

**Пример:**

“У вас есть CSV с событиями в мобильном приложении ( `user_id` , `event_type` , `timestamp` , `screen` ).

Нужно: ”

- Посчитать, сколько пользователей дошли до экрана оплаты.
- Найти «узкие места» в воронке (где больше всего отток).
- Сравнить конверсию до и после релиза новой версии.

# Кому нужно обрабатывать и анализировать наборы данных?

---

## 2. Разработчики

**Задача:** отладка, миграции, генерация тестовых данных.

**Пример:**

“После релиза приложения начали приходить жалобы на дубли заказов.

Логи сохранены в `orders_log.csv`.

Нужно: ”

- Найти все `order_id`, встречающиеся более одного раза.
- Проверить, не нарушена ли целостность данных (например, отрицательные суммы).
- Сгенерировать SQL-скрипт для исправления.

# Кому нужно обрабатывать и анализировать наборы данных?

---

## 3. Системные / DevOps-администраторы

Задача: анализ логов, мониторинг, аудит.

Пример:

“У вас есть CSV-файл с логами доступа к серверу ( `ip` , `timestamp` , `request` , `status` , `user_agent` ).  
Нужно: ”

- Найти IP-адреса с более чем 1000 запросов за час (возможный DDoS).
- Определить, сколько 5xx-ошибок было вчера.
- Экспортировать подозрительные запросы для расследования.

# Кому нужно обрабатывать и анализировать наборы данных?

---

## 4. Тестировщики (QA)

**Задача:** валидация данных, сравнение результатов.

**Пример:**

“После миграции базы данных выгрузили старые и новые данные в `users_old.csv` и `users_new.csv`.  
Нужно: ”

- Убедиться, что все пользователи перенесены.
- Найти расхождения в email или балансе.
- Проверить, что нет дубликатов в новой таблице.

# Кому нужно обрабатывать и анализировать наборы данных?

---

## 5. Менеджеры продукта / проекта

Задача: принятие решений на основе данных.

Пример:

“Получили отчёт в CSV: `feature_usage.csv` ( `user_id` , `feature_name` , `used_at` ).

Нужно: ”

- Оценить, сколько пользователей используют новую функцию.
- Понять, стоит ли вкладываться в её развитие или закрыть.
- Подготовить презентацию с графиками для команды.

# Кому нужно обрабатывать и анализировать наборы данных?

---

## 6. Финансисты / бухгалтеры

Задача: сверка, отчётность, выявление аномалий.

Пример:

“Есть выгрузка платежей `payments.csv` и список ожидаемых транзакций `expected.csv` .

Нужно: ”

- Найти пропущенные или лишние платежи.
- Посчитать итог по каждому контрагенту.
- Проверить, нет ли платежей с нулевой суммой.

# Кому нужно обрабатывать и анализировать наборы данных?

---

## 7. HR-специалисты

Задача: анализ текучести, эффективности найма.

Пример:

“Есть файл `employees.csv` с датами приёма/увольнения и отделами.

Нужно: ”

- Посчитать текучесть по отделам за последний квартал.
- Найти, сколько дней в среднем занимает закрытие вакансии.

# Задача обработки и анализа наборов данных

## **Общий паттерн**

Независимо от роли, типичные задачи сводятся к:

- Фильтрация («покажи только ошибки»),
- Агрегация («сколько всего?», «в среднем?»),
- Сравнение («было vs стало», «ожидаемо vs фактически»),
- Поиск аномалий (дубли, пустые значения, выбросы),
- Подготовка отчёта (таблица или CSV для коллег).

Именно поэтому лёгкие инструменты вроде SQLite или даже продвинутых возможностей Excel оказываются востребованы повсеместно — они позволяют быстро получить ответ, не дожидаясь помощи от аналитика или разработчика.

# CSV-файлы

```
1 address,postal_code,country,federal_district,region_type,region,area_type,area,city_type,city,
settlement_type,settlement,kladr_id,fias_id,fias_level,capital_marker,okato,oktmo,tax_office,
timezone,geo_lat,geo_lon,population,foundation_year
1 "Респ Адыгея, г Адыгейск",385200,Россия,Южный,Респ,Адыгея,, ,г,Адыгейск,, ,0100000200000,ccdfd496-
8108-4655-aadd-bd228747306d,4,0,79403000000,79703000001,0107,UTC+3,44.878414,39.190289,12689,1969
2 г Майкоп,385000,Россия,Южный,Респ,Адыгея,, ,г,Майкоп,, ,0100000100000,8cfbe842-e803-49ca-9347-
1ef90481dd98,4,2,79401000000,79701000001,0105,UTC+3,44.6098268,40.1006606,144055,1857
3 г Горно-Алтайск,649000,Россия,Сибирский,Респ,Алтай,, ,г,Горно-Алтайск,, ,0400000100000,0839d751-
b940-4d3d-afb6-5df03fdd7791,4,2,84401000000,84701000,0400,UTC+7,51.9581028,85.9603235,62861,1830
4 "Алтайский край, г Алейск",658125,Россия,Сибирский,край,Алтайский,, ,г,Алейск,, ,2200000200000,
ae716080-f27b-40b6-a555-cf8b518e849e,4,0,01403000000,01703000,2201,UTC+7,52.4922513,82.7793606,
28528,1913
5 г Барнаул,656000,Россия,Сибирский,край,Алтайский,, ,г,Барнаул,, ,2200000100000,d13945a8-7017-46ab-
b1e6-ede1e89317ad,4,2,01401000000,01701000,2200,UTC+7,53.3479968,83.7798064,635585,1730
6 "Алтайский край, г Белокуриха",659900,Россия,Сибирский,край,Алтайский,, ,г,Белокуриха,, ,
2200000300000,e4edca96-9b86-4cac-8c7f-cc93d9ba4cd1,4,0,01404000000,01704000001,2204,UTC+7,51.
996152,84.9839604,15072,1846
7 "Алтайский край, г Бийск",659300,Россия,Сибирский,край,Алтайский,, ,г,Бийск,, ,2200000400000,
52f876f6-cb1d-4f23-a22f-b692609fc1e0,4,0,01405000000,01705000001,2204,UTC+7,52.5393864,85.
2138453,203826,1709
8 "Алтайский край, г Горняк",658420,Россия,Сибирский,край,Алтайский,р-н,Локтевский,г,Горняк,, ,
2202700100000,094b3627-2699-4782-8492-4d82aac71958,4,1,01225501000,01625101,2209,UTC+7,50.
9979622,81.4643358,13040,1942
9 "Алтайский край, г Заринск",659100,Россия,Сибирский,край,Алтайский,, ,г,Заринск,, ,2200001100000,
142e04ef-dec1-44fa-b553-fac215764374,4,0,01406000000,01706000001,2208,UTC+7,53.7063476,84.
9315081,47035,1748
10 "Алтайский край, г Змеиногорск",658480,Россия,Сибирский,край,Алтайский,р-н,Змеиногорский,г,
Змеиногорск,, ,2201500100000,e7001b8f-d104-4873-96d4-66339f5e530a,4,1,01214501000,01614101,2209,
UTC+7,51.1581094,82.1872547,10569,1736
11 "Алтайский край, г Камень-на-Оби",658700,Россия,Сибирский,край,Алтайский,р-н,Каменский,г,Камень-
на-Оби,, ,2201800100000,810ca9c7-f10e-4def-9c48-f0aa83168ca7,4,1,01216501000,01616101001,2207,UTC+
7,53.7913974,81.3545053,41787,1751
12 "Алтайский край, г Новоалтайск",658041,Россия,Сибирский,край,Алтайский,, ,г,Новоалтайск,, ,
2200000800000,aa288d9f-4b2a-42a6-97f0-3502dddafa383,4,0,01413000000,01713000001,2208,UTC+7,53.
4119759,83.9311069,73134,1736
```

CSV (Comma-Separated Values) — один из самых распространённых форматов для хранения и обмена табличными данными.

Его популярность объясняется сочетанием простоты, универсальности и практичности.

## 1. Простота и читаемость

- CSV — это обычный текстовый файл.
- Каждая строка = одна запись (строка таблицы).
- Поля разделены запятыми (или другими разделителями: `;`, `\t`).
- Заголовки (имена колонок) часто идут в первой строке.

Пример:

csv

```
1 name,age,city
2 Alice,30,New York
3 Bob,25,London
```

## 2. Универсальная поддержка

CSV понимают практически все инструменты для работы с данными:

- Электронные таблицы: Excel, Google Sheets, LibreOffice Calc.
  - Языки программирования: Python ( `csv` , `pandas` ), R, Julia, JavaScript.
  - СУБД: PostgreSQL, MySQL, SQLite, SQL Server — все умеют импортировать CSV.
  - BI-системы: Power BI, Tableau, Looker.
  - Командная строка: `awk` , `cut` , `sort` , `grep` , `jq` (с преобразованием).
- Это делает CSV «универсальным языком обмена» между разными системами.

## 3. Лёгкость и компактность

- Нет метаданных, форматирования, стилей — только «голые» данные.
- Файлы легко сжимаются (например, `.csv.gz`).
- Подходит для передачи даже по медленным каналам связи.

## 4. Независимость от ПО и лицензий

- CSV — открытый, не проприетарный формат.
- Не требует установки дорогих программ (в отличие от `.xlsx`, который «родной» для Microsoft Excel).
- Может использоваться в open-source проектах без юридических ограничений.

→ Это критично для научных данных, государственных выгрузок, open data инициатив.

## 5. Простота генерации и парсинга

- Любой скрипт может записать строку в CSV без сложных библиотек.
- Парсинг тривиален: разделить строку по запятой (с учётом кавычек и экранирования).
- Даже базы данных могут экспортировать результаты запросов напрямую в CSV.

## 6. Подходит для машинной обработки

- Линейная структура → легко читать построчно, не загружая весь файл в память.
- Идеален для ETL-процессов, пайплайнов обработки, потоковой обработки.
- Хорошо интегрируется с командной строкой (Unix-философия: «делай одно дело хорошо»).

## Когда CSV не подходит?

- Данные иерархические (JSON, XML лучше).
- Требуется строгая типизация и схема (Parquet, Avro, Arrow).
- Нужна высокая производительность чтения/записи (Parquet, HDF5).
- Многотабличная структура (лучше архив с несколькими CSV или SQLite-файл).

Но даже в этих случаях CSV часто используется как **промежуточный** или **экспортный формат** из-за своей простоты.

---

**Вывод:** CSV — это язык данных по умолчанию. Он выигрывает не за счёт функциональности, а за счёт максимальной совместимости и минимального барьера для использования.

# **Инструменты для работы с CSV-файлами**

# SQLite как инструмент для работы с CSV

SQLite — это **встраиваемая СУБД без сервера**, которая может:

- Импортировать CSV-файлы в таблицы.
- Выполнять полноразмерные SQL-запросы (включая CTE, оконные функции, JOIN'ы).
- Экспортировать результаты обратно в CSV.
- Работать с **миллионами строк** даже на слабом железе.
- Не требовать установки сервера или сложной настройки.

# SQLite как инструмент для работы с CSV

Пример рабочего процесса:

```
bash
1 # 1. Создаём базу в памяти или на диске
2 sqlite3 data.db
3
4 # 2. Внутри SQLite:
5 .mode csv
6 .import employees.csv employees
7
8 # 3. Выполняем запрос
9 SELECT department, AVG(salary)
10 FROM employees
11 WHERE hire_date > '2020-01-01'
12 GROUP BY department;
13
14 # 4. Экспортируем результат
15 .output report.csv
16 SELECT ...
```

💡 Начиная с SQLite 3.38.0 (2022), появилась поддержка прямого чтения CSV через виртуальные таблицы ( `csv` extension), что позволяет работать с CSV без импорта: "

```
sql
1 CREATE VIRTUAL TABLE emp USING csv(filename='employees.csv', header=yes);
2 SELECT * FROM emp WHERE salary > 50000;
```

# Практическое применение SQL

## Инструменты для работы с наборами данных

Технология	Продукт	Достоинства	Недостатки
Электронные таблицы	Microsoft Excel	GUI, простота и наглядность. Хорошо подходит для простых разовых задач.	Тяжело работать со связанными данными. Данные не отделены от представления. Слабая повторяемость.
Скриптовый язык программирования	Python + Pandas (Jupyter Notebook)	Мощные возможности. Наглядная визуализация.	Необходимо изучать ЯП и специфические методы библиотек.
Язык SQL	SQLite	Декларативный язык специально для обработки данных. Простота установки.	Нет стандартных средств визуализации

# Когда что выбрать?

## ✓ Используй SQLite / SQL, если:

- Данные табличные и структурированные.
- Нужно быстро выполнить **агрегацию, фильтрацию, соединение нескольких CSV**.
- Требуется **повторяемый, документированный анализ** (запрос = отчёт).
- Работаешь в терминале или автоматизируешь через bash.
- Не хочешь устанавливать Python или тяжёлые зависимости.

“Пример: «Найти всех клиентов из файла `orders.csv`, у которых сумма заказов > 1000, используя данные из `customers.csv`» — делается двумя `IMPORT` и одним `JOIN` .”

# Когда что выбрать?

## ✅ Используй электронные таблицы, если:

- Данные небольшие (< 100 тыс. строк).
- Нужна **быстрая визуальная проверка**, ручная правка, диаграммы.
- Анализ разовый и интерактивный.
- Работаешь с нетехническими коллегами.

“Но: при обновлении данных всё приходится переделывать вручную.”

---

## ✅ Используй Python (pandas и др.), если:

- Нужна **сложная трансформация**: парсинг текста, машинное обучение, API-вызовы.
- Данные **неструктурированные** или «грязные».
- Требуется **визуализация**, статистика, интеграция с другими системами.
- Анализ — часть большого pipeline (ETL, веб-приложение и т.п.).

“Пример: очистка email-адресов, кластеризация клиентов, прогноз спроса.”

## 💡 Практический совет: гибридный подход

Часто лучше комбинировать:

1. **SQLite** — для первичной очистки, фильтрации, агрегации.
2. **Python/pandas** — для дальнейшего анализа или ML.
3. **Таблицы** — для презентации финального результата.

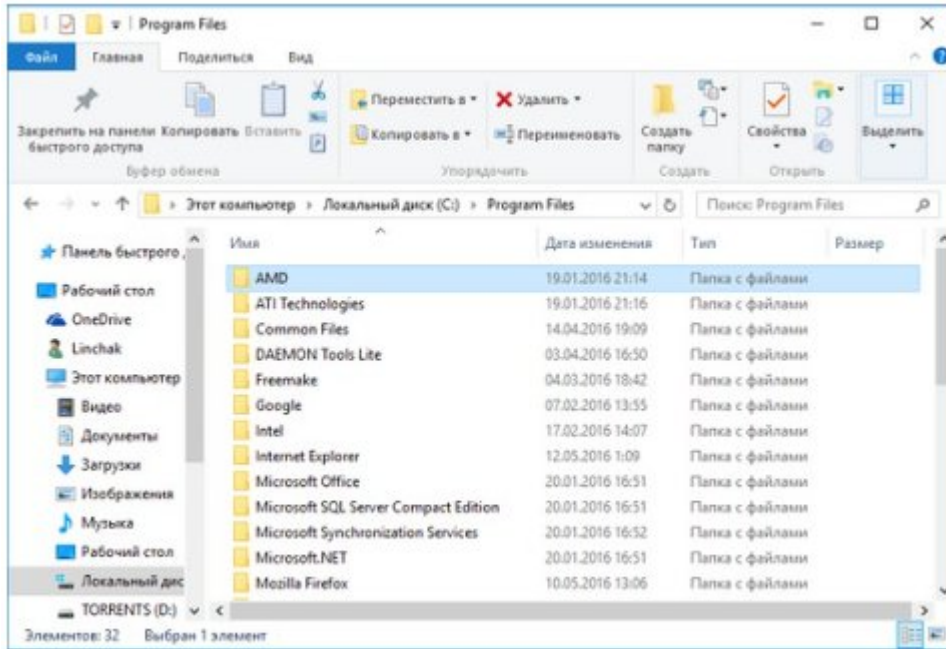
Например:

```
bash
```

```
1 sqlite3 -csv data.db "SELECT ... FROM big_file WHERE ..." > cleaned.csv
```

→ затем загрузить `cleaned.csv` в pandas для визуализации.

# GUI, CLI, скрипты и программы



```
C:\> copy d:\a.txt d:\b.txt
```

```
C:\python3.exe
```

```
>>> import shutil  
>>> shutil.copyfile("d:\\a.txt", "d:\\b.txt")
```

Командный интерфейс для управления

- Файловой системой
- Объектами ОС
- Базами данных и структурированными данными
- Текстом