

Лекция 7.

Язык SQL - продолжение

Условная логика

Условная логика в SQL

- Стандартный оператор CASE ... WHEN ... END.
- Встроенная функция, имитирующая if-then-else. Зависит от СУБД, в SQLite функция IIF().

Условный оператор CASE

Аналог if-else в других ЯП - проверяет условие и возвращает один из возможных вариантов.

Первая форма:	Вторая форма:
<pre>CASE WHEN условие_1 THEN возвращаемое_значение_1 ... WHEN условие_N THEN возвращаемое_значение_N [ELSE возвращаемое_значение] END</pre>	<pre>CASE проверяемое_значение WHEN сравниваемое_значение_1 THEN возвращаемое_значение_1 ... WHEN сравниваемое_значение_N THEN возвращаемое_значение_N [ELSE возвращаемое_значение] END</pre>

Можно использовать в операторах SELECT, INSERT, UPDATE, DELETE.

Условный оператор CASE

◆ Простой CASE (сравнение одного выражения)

```
sql
1 v CASE выражение
2     WHEN значение1 THEN результат1
3     WHEN значение2 THEN результат2
4     ...
5     ELSE результат_по_умолчанию
6 END
```

Пример: Классификация отделов

```
sql
1 v SELECT name,
2     department,
3     CASE department
4         WHEN 'IT'           THEN 'Технический'
5         WHEN 'Finance'     THEN 'Финансовый'
6         WHEN 'HR'          THEN 'Кадровый'
7         ELSE 'Прочие'
8     END AS dept_type
9 FROM employees;
```

Условный оператор CASE

◆ Поисковый **CASE** (условия с логикой)

```
sql
1 v CASE
2     WHEN условие1 THEN результат1
3     WHEN условие2 THEN результат2
4     ...
5     ELSE результат_по_умолчанию
6 END
```

Пример: Градация зарплат

```
sql
1 v SELECT name,
2     salary,
3     CASE
4         WHEN salary >= 90000 THEN 'Высокая'
5         WHEN salary >= 70000 THEN 'Средняя'
6         ELSE 'Низкая'
7     END AS salary_level
8 FROM employees;
```

SELECT: условный оператор CASE

Проверяет условие и возвращает один из возможных вариантов.

```
SELECT id, name AS 'Фамилия',  
       CASE city  
         WHEN 'Саранск' THEN 'Да'  
         ELSE 'Нет'  
       END 'Земляк'  
FROM customer;
```

```
SELECT id, name AS 'Фамилия',  
       CASE  
         WHEN rating >= 300 THEN 'Молодец'  
         WHEN rating >= 200 THEN 'Неплохо'  
         ELSE 'Старайся больше'  
       END 'Оценка по рейтингу'  
FROM customer;
```

Условный оператор CASE

Возвращаемым значение может быть результат подзапроса.

```
SELECT v.name,  
CASE  
  WHEN v.city='Саранск' THEN  
    (SELECT count(*) FROM `order` o  
     WHERE v.id=o.vendor_id)  
  ELSE 'Не из Саранска'  
END sales  
FROM vendor v;
```

4. 📌 Условная логика в других частях запроса

В WHERE :

```
sql
1 v SELECT *
2   FROM employees
3   WHERE CASE
4         WHEN department = 'Sales' THEN salary > 60000
5         WHEN department = 'IT'   THEN salary > 85000
6         ELSE salary > 70000
7   END;
```

В ORDER BY :

```
sql
1 v SELECT name, department
2   FROM employees
3   ORDER BY
4     CASE department
5       WHEN 'IT'      THEN 1
6       WHEN 'Finance' THEN 2
7       ELSE 3
8     END,
9   name;
```

Аналитические запросы и оконные функции

Зачем нужна аналитика данных?

Представьте: у вас есть таблица с данными — например, продажи, оценки студентов, посещаемость сайта, зарплаты сотрудников.

Если просто вывести эти данные, вы увидите что было, но не поймёте:

- Как меняется ситуация со временем? (рост или падение?)
- Кто лидирует, а кто отстаёт?
- Насколько типичен или аномален конкретный результат?
- Как один показатель соотносится с другими?

“💡 Аналитика — это не просто «данные», а «данные в контексте».”

Почему обычных SQL-запросов недостаточно?

Стандартные SQL-запросы с `SELECT`, `WHERE`, `GROUP BY` позволяют:

- Отфильтровать данные (`WHERE`)
- Сгруппировать и посчитать итоги (`GROUP BY + SUM`, `AVG` и т.д.)

Но есть проблема:

Когда вы используете `GROUP BY`, вы теряете детали. Например:

```
text
1  Исходные данные:
2  Иван – 100 руб.
3  Иван – 150 руб.
4  Мария – 200 руб.
5
6  Запрос с GROUP BY:
7  Иван – 250 руб. (итого)
8  Мария – 200 руб. (итого)
```

- Вы не видите, что у Ивана были две разные продажи — 100 и 150.
- Вы не можете сравнить первую продажу Ивана с его же второй продажей.
- Вы не можете посчитать, какая из продаж Ивана была лучше относительно его среднего.

Для чего нужны оконные функции?

“ Обычные агрегатные функции сворачивают данные.

 Аналитика часто требует сохранить детали, но добавить к ним контекст.”

Оконные функции позволяют добавить аналитический контекст к каждой строке, не теряя исходные данные.

Аналитические запросы и оконные функции

Пример 1: «Как каждый студент сдал экзамен по сравнению со средним по группе?»

Данные:

СТУДЕНТ	ОЦЕНКА
Аня	85
Боря	70
Вика	90

Обычный запрос:

→ Можно посчитать среднее: $(85 + 70 + 90) / 3 = 81.7$

→ Но чтобы показать это среднее рядом с каждой оценкой, нужна оконная функция!

Аналитические запросы и оконные функции

Результат с аналитикой:

СТУДЕНТ	ОЦЕНКА	СРЕДНЕЕ ПО ГРУППЕ	ОТКЛОНЕНИЕ
Аня	85	81.7	+3.3
Боря	70	81.7	-11.7
Вика	90	81.7	+8.3

- Теперь видно: Боря отстаёт, а Вика — лидер.
- И всё это без потери исходных строк!

Аналитические запросы и оконные функции

Пример 2: «Как менялись продажи магазина день за днём?»

Данные:

ДЕНЬ	ПРОДАЖИ
Пн	100
Вт	120
Ср	90

Вопросы:

- На сколько выросли продажи во вторник?
- Упали ли они в среду?

Обычный SQL не может сравнить строку с предыдущей.

Аналитические запросы и оконные функции

Обычный SQL не может сравнить строку с предыдущей.

Оконная функция `LAG()` — может!

Результат:

ДЕНЬ	ПРОДАЖИ	ПРОДАЖИ ВЧЕРА	РОСТ (%)
Пн	100	—	—
Вт	120	100	+20%
Ср	90	120	-25%

→ Видна динамика: рост, потом падение.

Аналитические запросы и оконные функции

Пример 3: «Кто лучший продавец в каждом регионе?»

Данные:

РЕГИОН	ПРОДАВЕЦ	ПРОДАЖИ
Север	Иван	200
Север	Анна	250
Юг	Борис	180
Юг	Катя	190

Хотим: присвоить ранг внутри каждого региона.

Аналитические запросы и оконные функции

Результат с `RANK() OVER (PARTITION BY Регион ORDER BY Продажи DESC)` :

РЕГИОН	ПРОДАВЕЦ	ПРОДАЖИ	МЕСТО
Север	Анна	250	1
Север	Иван	200	2
Юг	Катя	190	1
Юг	Борис	180	2

→ Без оконных функций пришлось бы писать сложные подзапросы или терять данные.

Аналитические запросы и оконные функции

“Оконные функции — это инструмент, который позволяет “видеть больше”, не теряя “то, что уже есть”.

Они добавляют к каждой строке контекст: среднее, ранг, предыдущее значение, накопительную сумму и т.д.

Это мост между сырыми данными и осмысленной аналитикой.”

Когда использовать оконные функции?

- Когда нужно сравнить строку с другими строками (в группе или во времени).
- Когда нужно сохранить детали, но добавить итоги или рейтинги.
- Когда важна динамика: рост, тренды, отклонения.

Они делают SQL не просто языком для извлечения данных, а языком для анализа и понимания этих данных.

Синтаксис и механизм выполнения оконных функций

Структура оператора SELECT

SELECT DISTINCT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT ...

Порядок выполнения основных этапов SQL-запроса

Хотя вы пишете запрос в порядке `SELECT → FROM → WHERE → GROUP BY → HAVING → ORDER BY → LIMIT`, на самом деле СУБД выполняет его в другом порядке:

1. `FROM` — загрузка и соединение таблиц
2. `WHERE` — фильтрация строк
3. `GROUP BY` — группировка строк (если есть)
4. `HAVING` — фильтрация групп
5. `SELECT` — выборка столбцов и вычисление выражений, включая оконные функции
6. `ORDER BY` — сортировка результата
7. `LIMIT` / `OFFSET` — ограничение числа строк

Логический порядок выполнения запроса

5 6 1 2 3 4 7 8
SELECT DISTINCT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT ...

1

FROM
– Источники данных:
таблицы, JOIN, CTE

5

SELECT
– Выбор и вычисление
столбцов
– Алиасы, скалярные
подзапросы, оконные
функции (OVER)

2

WHERE
– Фильтрация строк
– До группировки!
– Без агрегатов

6

DISTINCT
– Удаление дубликатов
по всем столбцам SELECT

3

GROUP BY
– Группировка строк
– Каждая группа → 1 строка

7

ORDER BY
– Сортировка результата
– Можно использовать
алиасы из SELECT

4

HAVING
– Фильтрация групп
– Можно использовать
агрегаты (AVG, SUM...)

8

LIMIT
– Ограничение числа
строк в финальном
результате

Окна данных и оконные функции

Можно разбить результирующий набор данных на **окна** - набор строк, в которых происходит вычисление функции.

Окна данных могут содержать от одной строки до всех строк из результирующего набора данных.

Результаты работы оконных функций добавляются к выборке как еще одно поле.

Синтаксис оконных функций

функция(выражение) **OVER** ([PARTITION BY выражения] [ORDER BY выражения])

1. ROW_NUMBER()

Нумерует строки **последовательно** (1, 2, 3, ...) в пределах окна.

→ Полезно для нумерации, выбора "топ-N" или устранения дубликатов.

2. RANK()

Присваивает **ранг с пропусками** при равенстве.

Пример: 100, 100, 90 → ранги: **1, 1, 3**

→ Используется для турнирных таблиц и рейтингов.

3. DENSE_RANK()

Присваивает ранг **без пропусков** при равенстве.

Пример: 100, 100, 90 → ранги: **1, 1, 2**

→ Удобно, когда важно сохранить последовательность рангов.

Синтаксис оконных функций

функция(выражение) **OVER** ([PARTITION BY выражения] [ORDER BY выражения])

4. `SUM() OVER (...)`

Считает **сумму** по окну.

→ Может использоваться как **итог по группе** или **накопительная сумма**, если добавить `ORDER BY`.

5. `AVG() OVER (...)`

Вычисляет **среднее значение** в рамках окна.

→ Позволяет сравнивать каждую строку со средним (например, «выше или ниже среднего по региону»).

6. `LAG(столбец, N)`

Возвращает значение из **предыдущей строки** (сдвинутое назад на N позиций).

→ Идеально для расчёта **изменений во времени**: рост/падение продаж, темпов прироста.

Синтаксис оконных функций

функция(выражение) **OVER** ([PARTITION BY выражения] [ORDER BY выражения])

7. LEAD(столбец, N)

Возвращает значение из **следующей строки** (сдвинутое вперёд на N позиций).

→ Полезно, когда нужно «заглянуть в будущее»: например, сравнить сегодня с завтрашним днём.

8. FIRST_VALUE() / LAST_VALUE()

Возвращает **первое** или **последнее** значение в окне.

→ Часто используется с **ORDER BY** для получения стартового/конечного значения в периоде (например, первая цена акции в месяце).

💡 Бонус: все эти функции используются с **OVER(...)**, где можно задать:

- **PARTITION BY** — разбить на группы (например, по региону или менеджеру),
- **ORDER BY** — задать порядок внутри группы,
- **ROWS/RANGE** — уточнить границы окна (например, «текущая строка + 2 предыдущие»).

Примеры оконных функций

```
SELECT *, ROW_NUMBER() OVER () AS number FROM customer;
```

```
SELECT *, ROW_NUMBER() OVER (ORDER BY rating DESC) AS rating_place FROM customer  
ORDER BY name;
```

```
SELECT *, RANK() OVER (ORDER BY rating DESC) AS rank FROM customer ORDER BY name;
```

```
SELECT *, MAX(rating) OVER () FROM customer;
```

```
SELECT name, city, rating, rating-AVG(rating) OVER () AS delta_rating FROM customer;
```

```
SELECT *, COUNT(*) OVER (PARTITION BY city) AS customers_in_city FROM customer;
```

Примеры на таблице `employees`

Вспомним нашу таблицу:

ID	NAME	DEPARTMENT	SALARY
1	Иван Петров	IT	90000
2	Мария Сидорова	IT	95000
3	Алексей Иванов	IT	85000
4	Ольга Кузнецова	HR	60000
5	Дмитрий Смирнов	HR	62000
6	Елена Попова	Finance	75000
7	Сергей Васильев	Finance	78000
8	Анна Морозова	Marketing	68000
9	Павел Новиков	Marketing	70000
10	Татьяна Лебедева	Sales	65000



1 Агрегатные функции как оконные

Пример: Средняя зарплата по отделу для каждого сотрудника

```
sql
1 SELECT
2     name,
3     department,
4     salary,
5     AVG(salary) OVER (PARTITION BY department) AS avg_dept_salary
6 FROM employees
7 ORDER BY department, salary;
```

Результат (частично):

NAME	DEPARTMENT	SALARY	AVG_DEPT_SALARY
Алексей Иванов	IT	85000	90000
Иван Петров	IT	90000	90000
Мария Сидорова	IT	95000	90000
Ольга Кузнецова	HR	60000	61000
Дмитрий Смирнов	HR	62000	61000
...

✓ Каждая строка сохраняется, но добавляется агрегированное значение по группе.

1 Агрегатные функции как оконные

Пример: Средняя зарплата по отделу для каждого сотрудника

```
sql
1 SELECT
2     name,
3     department,
4     salary,
5     AVG(salary) OVER (PARTITION BY department) AS avg_dept_salary
6 FROM employees
7 ORDER BY department, salary;
```

Результат (частично):

NAME	DEPARTMENT	SALARY	AVG_DEPT_SALARY
Алексей Иванов	IT	85000	90000
Иван Петров	IT	90000	90000
Мария Сидорова	IT	95000	90000
Ольга Кузнецова	HR	60000	61000
Дмитрий Смирнов	HR	62000	61000
...

✓ Каждая строка сохраняется, но добавляется агрегированное значение по группе.

2 Ранжирующие функции

а) ROW_NUMBER() — нумерация строк

```
sql
1 SELECT
2     name,
3     department,
4     salary,
5     ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rn
6 FROM employees;
```

→ Пронумерует сотрудников в пределах отдела по убыванию зарплаты.

Мария Сидорова — `m = 1` в IT, Сергей Васильев — `m = 1` в Finance и т.д.

б) RANK() — ранг с пропусками при совпадениях

```
sql
1 -- Допустим, добавим ещё одного сотрудника с зарплатой 95000 в IT
2 -- Тогда RANK() даст: 1, 1, 3 (а не 1, 1, 2)
```

с) DENSE_RANK() — ранг без пропусков

```
sql
1 SELECT
2     name,
3     salary,
4     DENSE_RANK() OVER (ORDER BY salary DESC) AS salary_rank
5 FROM employees;
```

→ Самая высокая зарплата — ранг 1, следующая уникальная — ранг 2 и т.д.

3 Накопительные (running) агрегаты

Пример: Накопительная сумма зарплат по компании (в порядке убывания зарплаты)

```
sql
1 SELECT
2     name,
3     salary,
4     SUM(salary) OVER (ORDER BY salary DESC ROWS UNBOUNDED PRECEDING) AS running_
5 FROM employees;
```

- **ROWS UNBOUNDED PRECEDING** = «от первой строки до текущей».
- Результат: каждая строка показывает, сколько денег «накопилось» до неё (включительно).

4 Сравнение с предыдущей/следующей строкой

Пример: Разница зарплаты с предыдущим сотрудником в отделе

```
sql
1 SELECT
2     name,
3     department,
4     salary,
5     LAG(salary) OVER (PARTITION BY department ORDER BY salary) AS prev_salary,
6     salary - LAG(salary) OVER (PARTITION BY department ORDER BY salary) AS diff
7 FROM employees
8 ORDER BY department, salary;
```

- `LAG()` — значение из предыдущей строки.
- `LEAD()` — значение из следующей строки.

Для HR:

- Ольга (60000) → `prev_salary = NULL`
- Дмитрий (62000) → `prev_salary = 60000` , `diff = 2000`

5 Процентные доли и позиции

Пример: Доля зарплаты сотрудника в общей сумме по отделу

```
sql
1 SELECT
2     name,
3     department,
4     salary,
5     ROUND(
6         salary * 100.0 / SUM(salary) OVER (PARTITION BY department),
7         2
8     ) AS pct_of_dept
9 FROM employees;
```

→ Покажет, сколько процентов от фонда отдела получает каждый.

📌 Когда использовать оконные функции?

- ✓ Когда нужно сохранить детализацию, но добавить агрегат/ранг/сравнение.
 - ✓ Вместо коррелированных подзапросов (часто быстрее и чище).
 - ✓ Для расчёта скользящих средних, рангов, кумулятивных сумм, процентилей.
-

⚠ Поддержка в СУБД

Оконные функции поддерживаются в:

- PostgreSQL ✓
- MySQL 8.0+ ✓
- SQL Server 2012+ ✓
- Oracle ✓
- SQLite 3.25+ ✓
- BigQuery, Snowflake, Redshift и др. ✓

⌋ "x В MySQL < 8.0 и старых версиях SQLite — не поддерживаются."

Замена коррелированных подзапросов

Примеры на таблице `employees`

Вспомним нашу таблицу:

ID	NAME	DEPARTMENT	SALARY
1	Иван Петров	IT	90000
2	Мария Сидорова	IT	95000
3	Алексей Иванов	IT	85000
4	Ольга Кузнецова	HR	60000
5	Дмитрий Смирнов	HR	62000
6	Елена Попова	Finance	75000
7	Сергей Васильев	Finance	78000
8	Анна Морозова	Marketing	68000
9	Павел Новиков	Marketing	70000
10	Татьяна Лебедева	Sales	65000



Замена коррелированных подзапросов

“Задача: Найти сотрудников, чья зарплата выше средней по их отделу.”

 Исходный запрос с коррелированным подзапросом:

```
sql
1 SELECT e1.name, e1.department, e1.salary
2 FROM employees e1
3 WHERE e1.salary > (
4     SELECT AVG(e2.salary)
5     FROM employees e2
6     WHERE e2.department = e1.department
7 );
```

Этот запрос выполняет подзапрос для каждой строки — неэффективно при большом объёме данных.

✓ Решение с оконной (аналитической) функцией

Мы можем предварительно вычислить среднюю зарплату по отделу для каждой строки — с помощью `AVG() OVER (PARTITION BY department)` — и затем отфильтровать результат.

```
sql
1 SELECT name, department, salary
2 FROM (
3     SELECT
4         name,
5         department,
6         salary,
7         AVG(salary) OVER (PARTITION BY department) AS avg_dept_salary
8     FROM employees
9 ) emp_with_avg
10 WHERE salary > avg_dept_salary;
```

🔍 Как это работает:

1. Внутренний запрос:

- Для каждой строки вычисляется `avg_dept_salary` — средняя зарплата в её отделе.
- Все исходные строки сохраняются (никакого `GROUP BY`!).

2. Внешний запрос:

- Фильтрует только тех, у кого `salary > avg_dept_salary`.

Замена коррелированных подзапросов

 Пример результата (на основе нашей таблицы):

NAME	DEPARTMENT	SALARY	AVG_DEPT_SALARY
Мария Сидорова	IT	95000	90000
Дмитрий Смирнов	HR	62000	61000
Сергей Васильев	Finance	78000	76500
Павел Новиков	Marketing	70000	69000

→ Эти 4 сотрудника и будут в финальном результате.

✔ Преимущества оконного подхода:

КРИТЕРИЙ	КОРРЕЛИРОВАННЫЙ ПОДЗАПРОС	ОКОННАЯ ФУНКЦИЯ
Производительность	$O(N \times M)$ — медленно	$O(N \log N)$ или лучше — быстро
Читаемость	Сложнее понять логику	Чёткое разделение вычисления и фильтрации
Оптимизация СУБД	Труднее оптимизировать	Легко параллелится и кэшируется
Гибкость	Только фильтрация	Можно сразу добавить ранг, разницу и т.д.

Вывод

“Коррелированные подзапросы часто можно (и нужно) заменять оконными функциями, особенно когда речь идёт о сравнении со статистикой по группе (среднее, максимум, ранг и т.п.).”

Где и как выполнять аналитику?

Выбор между аналитикой в SQL (с оконными функциями) и выгрузкой в Excel зависит от целей, объёма данных, частоты анализа, требований к точности и производительности.

 **Когда** лучше использовать SQL с оконными функциями:

- У вас больше 100 000 строк.
- Анализ повторяется регулярно (ежедневно, еженедельно).
- Нужна точность (финансы, расчёты зарплат, метрики продукта).
- Вы работаете в команде (аналитики, инженеры, менеджеры).
- Данные чувствительны (персональные, финансовые).
- Вы строите отчёты или дашборды (через Power BI, Looker, Metabase и т.п.).

Где и как выполнять аналитику?

Выбор между аналитикой в SQL (с оконными функциями) и выгрузкой в Excel зависит от целей, объёма данных, частоты анализа, требований к точности и производительности.

✓ Когда допустимо использовать Excel:

- Данных мало (< 10 000 строк).
- Анализ разовый, исследовательский («а что, если...?»).
- Нужно быстро нарисовать график или показать коллеге.
- Вы не владеете SQL, но умеете в Excel.
- Данные уже агрегированы (итоги по месяцам, не сырые события).

Объединение результатов запросов

UNION

Объединение строк из нескольких результирующих наборов данных.

- Количество столбцов в каждом запросе одно и то же.
- Столбцы в каждом запросе имеют совместимые типы.
- Имена столбцов в первом запросе задают имена для всего объединения.
- ORDER BY применяется ко всему набору.
- GROUP BY и HAVING применяются к каждому подзапросу.

Вывести всех покупателей и продавцов с указанием их роли и города, где они живут. Сортировать по имени города и фамилии человека.

```
SELECT name, city, 'Продавец' AS 'Роль' FROM vendor
UNION
SELECT name, city, 'Покупатель' FROM customer
ORDER BY city, name;
```

UNION и UNION ALL

UNION убирает в результирующем наборе повторяющиеся строки.

UNION ALL оставляет все строки.

```
SELECT 1, 2      1 | 2
  UNION          ---
SELECT 1, 2      1 | 2
  UNION          3 | 4
SELECT 3, 4;
```

```
SELECT 1, 2      1 | 2
  UNION ALL      ---
SELECT 1, 2      1 | 2
  UNION ALL      1 | 2
SELECT 3, 4;     3 | 4
```

Представления (view)

Представления

Объект базы данных, являющийся результатом выполнения запроса к базе данных, определенного с помощью оператора SELECT, в момент обращения к представлению.

- Упрощение запросов.
- Гибкая настройка прав доступа к данным (права даются не на таблицу, а на представление).
- Разделение логики хранения данных и программного обеспечения.

CREATE VIEW имя [столбцы]

AS выборка

```
CREATE VIEW orders_with_names
```

```
AS SELECT `order`.order_date, customer.name AS customer_name,  
vendor.name AS vendor_name, `order`.summa
```

```
FROM `order`, customer, vendor
```

```
WHERE `order`.customer_id = customer.id AND `order`.vendor_id =  
vendor.id;
```