

ПРАКТИЧЕСКАЯ РАБОТА №4. АНАЛИТИЧЕСКИЕ ЗАПРОСЫ: ОКОННЫЕ ФУНКЦИИ И ПОСТРОЕНИЕ СВОДНЫХ ТАБЛИЦ

Цель работы:

Целью данной практической работы является формирование у студентов углубленных навыков работы со сложными аналитическими запросами в СУБД PostgreSQL.

По завершении работы студент должен уметь:

- Формировать практические навыки применения оконных функций для решения аналитических задач, которые выходят за рамки стандартной агрегации с использованием GROUP BY.
- Освоить синтаксис предложения OVER(), включая использование PARTITION BY для логического разделения наборов данных на независимые группы (разделы) и ORDER BY для установления порядка строк внутри каждого раздела, что является основой для многих вычислений.
- Научиться проводить агрегатные вычисления (например, находить среднее значение по группе), не теряя при этом детализацию исходных строк, что является ключевым отличием и преимуществом оконных функций перед стандартной группировкой GROUP BY.
- Изучить и научиться применять на практике различные категории оконных функций для решения конкретных бизнес-задач:
 - **Ранжирующие функции** (ROW_NUMBER(), RANK(), DENSE_RANK()) для нумерации записей, определения рейтинга товаров или клиентов.

- **Агрегатные оконные функции** (SUM(), AVG(), COUNT()) для вычисления нарастающих итогов (кумулятивных сумм) и скользящих средних, что часто требуется в финансовом анализе и анализе временных рядов.
- **Функции смещения** (LAG(), LEAD()) для доступа к данным из предыдущих или последующих строк, что необходимо для сравнения показателей во времени (например, продажи текущего месяца с предыдущим).
- Получить практический опыт в преобразовании данных из «длинного» формата в «широкий» (построение сводных таблиц или PIVOT-преобразование) для создания наглядных отчетов. Освоить два ключевых подхода: с использованием условной агрегации (CASE) и с применением специализированной функции crosstab.

Постановка задачи:

Для выполнения практической работы необходимо последовательно выполнить четыре задачи, используя собственную базу данных. Все примеры в данном документе основаны на демонстрационной базе данных «Аптека», содержащей таблицы manufacturers (производители), medicines (лекарства) и sales (продажи).

Ваша задача — адаптировать каждую из поставленных задач к логической структуре и предметной области вашей базы данных. Приведенные ниже формулировки и последующие примеры кода служат шаблоном для понимания, какой тип аналитического запроса требуется составить.

Задание №1: использование ранжирующих функций

Для каждой основной «родительской» сущности в вашей БД (*например, производитель, категория товара, автор*) определить **три** наиболее значимых по некоторому **числовому признаку** дочерних сущности (*например, три самых дорогих товара, три самые популярные книги по количеству продаж*).

В результирующей таблице должны быть указаны идентификатор группы, идентификатор дочерней сущности, её числовой признак и ранг. Для расчёта ранга использовать функцию **RANK()** или **DENSE_RANK()**.

Задание №2: использование агрегатных оконных функций

Для ключевой сущности, имеющей **транзакции во времени** (например, товар, услуга), рассчитать **нарастающий итог** (кумулятивную сумму) по некоторому показателю (например, объем продаж, количество заказов) с разбивкой по временным периодам (месяцам или годам).

Отчёт должен содержать идентификатор сущности, временной период, сумму за период и кумулятивную сумму с начала наблюдений.

Задание №3: использование функции смещения

Провести сравнительный анализ общих показателей **по периодам**.

Для **каждого периода** (например, месяца), начиная со второго, необходимо вывести **общий показатель** за **текущий** период и аналогичный показатель за **предыдущий** период в одной строке. Это позволит наглядно оценить динамику.

Необходимо использовать функцию **LAG()**.

Задание №4: построение сводной таблицы

Создать сводный отчет, который агрегирует некоторый числовой показатель для основной сущности по категориям, представленным в виде столбцов.

Например, показать общую сумму продаж для каждого товара по кварталам года.

Строки в отчете должны представлять основные сущности, а столбцы — категории. Задачу необходимо **решить двумя способами**:

1. С использованием условной агрегации (комбинация **SUM** и **CASE**).
2. С использованием функции **crosstab** из расширения **tablefunc**.

Каждый SQL-запрос **сопроводить комментарием**, объясняющим его назначение и логику работы с учетом специфики вашей базы данных.

Подготовка базы данных:

Перед выполнением заданий убедитесь, что структура и наполнение вашей базы данных соответствуют требованиям. При необходимости дополните таблицы данными или внесите изменения в структуру.

Задание №1 (ранжирование)

В **базе данных** должны быть как минимум две **таблицы** с отношением «**один-ко-многим**» (например, *Категории и Товары*).

В «**дочерней**» **таблице** (Товары) обязательно должен присутствовать **числовой столбец**, по которому можно проводить ранжирование (например, *цена, рейтинг, количество_на_складе*).

Для наглядного результата необходимо создать **не менее 3 записей** в «**родительских**» **таблицах**, каждой из которых будет соответствовать **не менее 3-4 дочерних записей**.

Задания №2, №3, №4 (анализ по времени и сводные таблицы)

Эти задачи требуют наличия в базе данных таблицы с **транзакционными** или **историческими данными** (таблицы *Продажи* или *Заказы*).

В этой таблице **должен быть** столбец с типом данных **DATE** или **TIMESTAMP** (например, *дата_продажи, дата_заказа*).

Также **необходим числовой столбец** для проведения вычислений (например, *сумма_продажи, количество*).

Для корректного выполнения заданий необходимо, чтобы в этой таблице было **не менее 10-15 записей**, причем даты в этих записях должны охватывать несколько **разных периодов** (например, *несколько месяцев или кварталов одного года*). Это позволит корректно рассчитать нарастающие итоги, сравнить периоды и построить информативную сводную таблицу.

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

Введение

В этом разделе на примере демонстрационной базы данных «Аптека» показаны принципы работы с оконными функциями и шаблоны запросов для решения поставленных задач.

Вам необходимо изучить эти примеры и, по их аналогии, составить собственные запросы, которые будут работать со структурой вашей базы данных.

Ниже приводятся таблицы, используемые для построения запросов.

Таблица 1. Таблица *manufacturers* (Производители)

	123 manufacturer_id	A-Z manufacturer_name	A-Z country
1	1	ООО "Фармстандарт"	Россия
2	2	Bayer AG	Германия

Таблица 2. Таблица *medicines* (Лекарства)

	123 id	A-Z name	123 quantity_in_stock	123 price	production_date	expiration_date	123 manufacturer_id
1	1	Парацетамол	200	50,5	2025-07-10	2028-07-10	1
2	2	Аспирин	150	120	2025-07-12	2027-07-12	2
3	3	Ибупрофен	100	85	2025-07-11	2028-07-11	1
4	4	Витамин С	300	250	2025-07-10	2027-07-10	2

Таблица 3. Таблица *monthly_revenue* (Доход по месяцам)

	123 year	123 month	123 income	123 expense
1	2 024	1	94 000	82 000
2	2 024	2	94 000	75 000
3	2 024	3	94 000	104 000
4	2 024	4	100 000	94 000
5	2 024	5	100 000	99 000
6	2 024	6	100 000	105 000
7	2 024	7	100 000	95 000
8	2 024	8	100 000	110 000
9	2 024	9	104 000	104 000
10	2 024	10	104 000	100 000
11	2 024	11	104 000	98 000
12	2 024	12	104 000	106 000

Таблица 4. Таблица *pharmacist_ratings* (Рейтинг фармацевтов)

	A-Z name	A-Z month	123 rating
1	Петя	январь	4
2	Петя	февраль	3
3	Петя	март	5
4	Петя	апрель	4
5	Маша	январь	4
6	Маша	февраль	3
7	Маша	март	5
8	Маша	апрель	3

Таблица 5. Таблица *regional_sales* (Региональные продажи)

	A-Z region	A-Z medicine	123 amount
1	Север	Парацетамол	1 500
2	Север	Аспирин	800
3	Юг	Парацетамол	1 200
4	Юг	Аспирин	1 000
5	Восток	Парацетамол	1 300
6	Восток	Аспирин	700

Таблица 6. Таблица *sales* (продажи)

	123 sale_id	123 medicine_id	sale_date	123 quantity_sold	123 amount
1	1	1	2025-07-15	30	1 515
2	2	2	2025-07-20	20	2 400
3	3	3	2025-07-22	15	1 275
4	4	1	2025-08-10	25	1 262,5
5	5	2	2025-08-18	35	4 200
6	6	4	2025-08-25	50	12 500
7	7	1	2025-09-05	40	2 020
8	8	2	2025-09-12	10	1 200
9	9	3	2025-09-19	20	1 700

Введение в оконные функции: анализ без потери деталей

Основное отличие и преимущество оконных функций от стандартной агрегации **GROUP BY** заключается в том, что они выполняют вычисления над набором строк, но при этом возвращают результат для каждой строки, не «схлопывая» их в одну.

Рассмотрим простую задачу на примере БД «Аптека»:

«Для каждого лекарства вывести его цену и среднюю цену всех лекарств его производителя».

Попытка решить эту задачу с помощью **GROUP BY** приведет к потере информации об *отдельных* лекарствах, так как **GROUP BY** сгруппирует все строки по производителю и вернет только *одну строку* с агрегированным значением для каждого из них.

Листинг 1. Категоризация лекарств по цене

```
-- Для каждого лекарства выводим его название, цену,  
-- а также среднюю цену по всем лекарствам того же производителя.  
-- Оконная функция AVG(...) OVER(...) позволяет рассчитать среднее  
-- значение для группы строк (раздела окна), определенной в PARTITION BY,  
-- и вернуть это значение для каждой строки в разделе.  
SELECT  
    m.name AS "Название лекарства",  
    m.price AS "Цена",  
    man.manufacturer_name AS "Производитель",  
    -- Оконная функция: вычисляет среднюю цену  
    -- в рамках раздела окна (PARTITION BY), заданного производителем.  
    AVG(m.price) OVER (PARTITION BY m.manufacturer_id) AS "Средняя цена  
у производителя"  
FROM  
    medicines AS m  
JOIN  
    manufacturers AS man ON m.manufacturer_id = man.manufacturer_id;
```

В этом запросе **PARTITION BY m.manufacturer_id** делит все лекарства на группы (окна) по производителю.

Затем функция **AVG(m.price)** вычисляется для каждой группы отдельно, и результат этого вычисления добавляется в каждую строку соответствующей группы, сохраняя при этом полную детализацию по каждому лекарству.

	A-Z Название лекарства	123 Цена	A-Z Производитель	123 Средняя цена у производителя
1	Парацетамол	50,5	ООО "Фармстандарт"	67,75
2	Ибупрофен	85	ООО "Фармстандарт"	67,75
3	Аспирин	120	Bayer AG	185
4	Витамин С	250	Bayer AG	185

Рисунок 1 – Результат запроса с использованием PARTITION BY

1. Ранжирование записей (**ROW_NUMBER**, **RANK**, **DENSE_RANK**)

Ранжирующие функции присваивают каждой строке в окне определенный **ранг** в соответствии с порядком, заданным в предложении **ORDER BY**.

- **ROW_NUMBER()**: присваивает уникальный последовательный номер каждой строке. Не учитывает дубликаты.
- **RANK()**: присваивает ранг на основе значения. Строки с одинаковыми значениями получают одинаковый ранг. После группы с одинаковым рангом следующий ранг будет пропущен (например, 1, 2, 2, 4).
- **DENSE_RANK()**: работает аналогично **RANK()**, но не пропускает ранги после группы с одинаковыми значениями (например, 1, 2, 2, 3).

Таким образом, если нужно строго «топ-3 без совпадений» — используйте **ROW_NUMBER**, если допускаются «все, кто разделил 3-е место» — **DENSE_RANK**.

Найдём три самых дорогих лекарства у каждого производителя.

Для решения этой задачи нам нужно сначала проранжировать все лекарства по цене внутри группы каждого производителя, а затем отобрать только те, у которых ранг меньше или равен 3.

Использовать оконную функцию напрямую в **WHERE** нельзя из-за логического порядка выполнения запроса (оконные функции вычисляются после **WHERE**). Поэтому необходимо использовать подзапрос или общее табличное выражение (**CTE**).

Листинг 2. Пример – поиск трёх самых дорогих лекарств у каждого производителя

```
WITH RankedMedicines AS (  
    SELECT  
        m.name AS medicine_name,  
        m.price,  
        man.manufacturer_name,  
        DENSE_RANK() OVER (  

```



```

        PARTITION BY m.manufacturer_id
        ORDER BY m.price DESC
    ) AS price_rank
FROM
    medicines AS m
JOIN
    manufacturers AS man ON m.manufacturer_id = man.manufacturer_id
)
SELECT
    manufacturer_name AS "Производитель",
    medicine_name AS "Название лекарства",
    price AS "Цена",
    price_rank AS "Ранг"
FROM
    RankedMedicines
WHERE
    price_rank <= 3
ORDER BY
    manufacturer_name, price_rank;

```

На **первом шаге** (внутри **CTE**) мы ранжируем все лекарства.

Ранжируем лекарства по убыванию цены (самые дорогие сначала) в пределах каждого производителя (**PARTITION BY**).

При этом используем **DENSE_RANK**, чтобы лекарства с одинаковой ценой получили одинаковый ранг без пропусков.

На **втором шаге** отбираем из результатов ранжирования только те строки, где ранг не превышает 3.

	A-Z Производитель	A-Z Название лекарства	123 Цена	123 Ранг
1	Bayer AG	Витамин С	250	1
2	Bayer AG	Аспирин	120	2
3	ООО "Фармстандарт"	Ибупрофен	85	1
4	ООО "Фармстандарт"	Парацетамол	50,5	2

Рисунок 2 – Результат запроса

2. Агрегатные вычисления в окне: нарастающие итоги

Стандартные агрегатные функции, такие как **SUM()**, **AVG()**, **COUNT()**, могут использоваться как оконные.

В сочетании с **ORDER BY** внутри **OVER()** они позволяют вычислять нарастающие итоги или скользящие средние.

По умолчанию, если указан **ORDER BY**, **окно** (или «фрейм») включает **все строки** от начала раздела до текущей строки.

Листинг 3. Кумулятивные итоги по месяцам

```
WITH MonthlySales AS (  
    SELECT  
        m.name AS medicine_name,  
        DATE_TRUNC('month', s.sale_date)::DATE AS sale_month,  
        SUM(s.amount) AS monthly_amount  
    FROM  
        sales AS s  
    JOIN  
        medicines AS m  
    ON s.medicine_id = m.id  
    GROUP BY  
        m.name, sale_month  
)  
SELECT  
    medicine_name AS "Название лекарства",  
    sale_month AS "Месяц продажи",  
    monthly_amount AS "Сумма за месяц",  
    SUM(monthly_amount) OVER (  
        PARTITION BY medicine_name  
        ORDER BY sale_month  
    ) AS "Нарастающий итог"  
FROM  
    MonthlySales  
ORDER BY  
    medicine_name, sale_month;
```

Сначала **агрегируем продажи** по месяцам для каждого лекарства.

Затем используем **оконную** функцию для расчета **нарастающего итога**.

Вычисляем кумулятивную (нарастающую) сумму продаж.

PARTITION BY medicine_name - расчёт ведётся **независимо** для каждого лекарства.

ORDER BY sale_month - определяет порядок, в котором суммируются значения.

	ⓘ A-Z Название лекарства ▾	🕒 Месяц продажи ▾	123 Сумма за месяц ▾	123 Нарастающий итог ▾
1	Аспирин	2025-07-01	2 400	2 400
2	Аспирин	2025-08-01	4 200	6 600
3	Аспирин	2025-09-01	1 200	7 800
4	Витамин С	2025-08-01	12 500	12 500
5	Ибупрофен	2025-07-01	1 275	1 275
6	Ибупрофен	2025-09-01	1 700	2 975
7	Парацетамол	2025-07-01	1 515	1 515
8	Парацетамол	2025-08-01	1 262,5	2 777,5
9	Парацетамол	2025-09-01	2 020	4 797,5

Рисунок 3 – Результат запроса

3. Анализ последовательностей: функции смещения LAG и LEAD

Функции смещения позволяют получить доступ к данным из других строк в пределах того же окна.

- **LAG**(column, offset, default): возвращает значение из столбца **column** строки, которая находится на **offset** позиций **раньше** текущей.
- **LEAD**(column, offset, default): возвращает значение из столбца **column** строки, которая находится на **offset** позиций **позже** текущей.

Здесь:

offset — смещение, по умолчанию равно **1**.

default — это значение, которое будет возвращено, если смещенная строка выходит за пределы окна (например, для первой строки нет предыдущей).

Листинг 4. Сравнение продаж с предыдущим месяцем

```
WITH TotalMonthlySales AS (
    SELECT
        DATE_TRUNC('month', sale_date)::DATE AS sale_month,
        SUM(amount) AS total_amount
    FROM
        sales
    GROUP BY
        sale_month
)
```

```

SELECT
    sale_month AS "Месяц",
    total_amount AS "Продажи за текущий месяц",
    LAG(total_amount, 1, 0) OVER (ORDER BY sale_month) AS "Продажи за
    предыдущий месяц"
FROM
    TotalMonthlySales
ORDER BY
    sale_month;

```

В начале, как и в прошлой задаче, **агрегируем** все продажи **по месяцам**.

После этого используем функцию **LAG** для получения данных за **прошлый месяц**.

Функция **LAG**(total_amount, 1, 0) берет значение из столбца **total_amount** – из строки, предшествующей текущей (смещение 1).

ORDER BY sale_month гарантирует, что строки упорядочены хронологически.

Если **предыдущей строки нет** (для самого первого месяца), **вернётся 0**.

	Месяц	Продажи за текущий месяц	Продажи за предыдущий месяц
1	2025-07-01	5 190	0
2	2025-08-01	17 962,5	5 190
3	2025-09-01	4 920	17 962,5

Рисунок 4 – Результат запроса

4. Построение сводных таблиц (Pivoting)

Pivoting — это процесс преобразования данных из формата, где каждая характеристика объекта находится в отдельной строке, в формат, где эти характеристики становятся столбцами. Это часто используется для создания отчетов.

Перед выбором метода стоит учесть их особенности:

- Подход с **CASE** является частью **стандарта ANSI SQL**, что делает его переносимым между разными СУБД. Он интуитивно понятен и хорошо

подходит, когда количество и названия итоговых столбцов (категорий) заранее известны и фиксированы.

- В свою очередь, **crosstab** — это мощное, но **специфичное** для **PostgreSQL** расширение. Оно более эффективно и лаконично, когда количество итоговых столбцов велико или может меняться динамически, но требует **предварительной установки расширения** и имеет более сложный синтаксис.

4.1 Условная агрегация (SUM & CASE)

В методе с использованием SUM и CASE, мы группируем данные по лекарствам, а для каждого квартала создаем отдельный столбец с помощью CASE, который возвращает сумму продаж, только если дата продажи **попадает в нужный квартал**, и **0 в противном случае**.

Листинг 5. Метод с SUM и CASE

```
SELECT
    m.name AS "Название лекарства",
    SUM(CASE WHEN EXTRACT(QUARTER FROM s.sale_date) = 1 THEN s.amount
    ELSE 0 END) AS "Q1",
    SUM(CASE WHEN EXTRACT(QUARTER FROM s.sale_date) = 2 THEN s.amount
    ELSE 0 END) AS "Q2",
    SUM(CASE WHEN EXTRACT(QUARTER FROM s.sale_date) = 3 THEN s.amount
    ELSE 0 END) AS "Q3",
    SUM(CASE WHEN EXTRACT(QUARTER FROM s.sale_date) = 4 THEN s.amount
    ELSE 0 END) AS "Q4"
FROM
    sales AS s
JOIN
    medicines AS m ON s.medicine_id = m.id
GROUP BY
    m.name
ORDER BY
    m.name;
```

В начале производим группировку по названию лекарства, после чего для каждого квартала вычисляем сумму продаж с помощью условной агрегации.

Оператор **EXTRACT(QUARTER FROM sale_date)** извлекает номер квартала из даты.

	A-Z Название лекарства ▼	123 Q1 ▼	123 Q2 ▼	123 Q3 ▼	123 Q4 ▼
1	Аспирин	0	0	7 800	0
2	Витамин С	0	0	12 500	0
3	Ибупрофен	0	0	2 975	0
4	Парацетамол	0	0	4 797,5	0

Рисунок 5 – Результат запроса

4.2 Метод с crosstab из расширения tablefunc.

Этот метод требует расширения **tablefunc**.

Если оно не установлено, выполните команду:

```
CREATE EXTENSION IF NOT EXISTS tablefunc;
```

Функция **crosstab** принимает на вход **SQL-запрос**, который **должен возвращать** три столбца:

- **Идентификатор строки** (в нашем случае — название лекарства).
- **Категория**, которая станет названием столбца (квартал).
- **Значение для ячейки**.

Листинг 6. Метод с crosstab

```
SELECT * FROM crosstab(  
    'SELECT  
        m.name,  
        EXTRACT(QUARTER FROM s.sale_date),  
        SUM(s.amount)  
    FROM sales AS s  
    JOIN medicines AS m ON s.medicine_id = m.id  
    GROUP BY m.name, EXTRACT(QUARTER FROM s.sale_date)  
    ORDER BY 1, 2',  
    'SELECT q FROM generate_series(1,4) AS q'
```

```
) AS ct("Название лекарства" TEXT, "Q1" NUMERIC, "Q2" NUMERIC, "Q3" NUMERIC, "Q4" NUMERIC);
```

В начале вызываем функцию **crosstab**, передавая ей исходный запрос и определяя структуру результирующей таблицы.

1 этап. Исходный запрос (source_sql) должен возвращать 3 столбца: идентификатор строки, категорию столбца и значение.

Важно, чтобы результат был **отсортирован по первым двум столбцам** (ORDER BY 1, 2).

2 этап. Запрос категорий (categories_sql). Возвращает уникальные значения, которые станут заголовками столбцов.

3 этап. Определение результирующей таблицы. Учтите, что столбцы должны соответствовать идентификатору строки и категориям из второго запроса, типы данных должны быть указаны корректно.

	A-Z Название лекарства ▼	123 Q1 ▼	123 Q2 ▼	123 Q3 ▼	123 Q4 ▼
1	Аспирин	[NULL]	[NULL]	7 800	[NULL]
2	Витамин С	[NULL]	[NULL]	12 500	[NULL]
3	Ибупрофен	[NULL]	[NULL]	2 975	[NULL]
4	Парацетамол	[NULL]	[NULL]	4 797,5	[NULL]

Рисунок 6 – Результат запроса

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Объясните принципиальное различие в результате работы запроса с **SUM(...) OVER (PARTITION BY...)** и запроса с **SUM(...)** и **GROUP BY**. В какой ситуации каждый из них предпочтителен?
2. В чем разница между функциями **RANK()** и **DENSE_RANK()** при наличии одинаковых значений в столбце, по которому идет сортировка? Приведите бизнес-сценарий, в котором использование **DENSE_RANK()** будет более корректным.
3. Почему нельзя использовать оконную функцию непосредственно в предложении **WHERE** (например, **WHERE price > AVG(price) OVER (...)**)?
Опишите, как можно обойти это ограничение, используя **CTE** или подзапрос.
4. Какие два основных SQL-запроса требуются в качестве аргументов для функции **crosstab**?
Каково назначение каждого из них?

КРАТКИЙ СПРАВОЧНЫЙ МАТЕРИАЛ

1. Оконные функции: вычисления без потери строк

1.1 Что такое оконные функции и зачем они нужны?

Оконные функции выполняют вычисления **над набором строк** (так называемым «окном»), которые связаны с текущей строкой, но, в отличие от **GROUP BY**, они **не схлопывают** строки, а возвращают результат для каждой из них. Это позволяет одновременно видеть и детальные данные, и агрегированные показатели.

Основной синтаксис выглядит так:

Листинг 7.

```
ФУНКЦИЯ() OVER (  
    PARTITION BY столбец_для_группировки  
    ORDER BY столбец_для_сортировки  
    [ROWS | RANGE | GROUPS] условие_фильтрации  
)
```

OVER(): ключевое слово, которое указывает, что это оконная функция.

PARTITION BY: необязательная часть. Делит набор строк на независимые группы (разделы или «окна»). Аналогично **GROUP BY**, но не сворачивает строки.

ORDER BY: необязательная часть. Задает порядок строк внутри каждого раздела. Критически важен для функций ранжирования и расчета нарастающих итогов.

[ROWS | RANGE | GROUPS]: фрейм. Дополнительная «рамка», движущаяся внутри «окна».

1.2 Сравнение: оконная функция Vs GROUP BY

Таблица 2. Функции для работы с датой и временем

Критерий	SUM(...) OVER (PARTITION BY ...)	SUM(...) с GROUP BY
----------	----------------------------------	---------------------

Основная цель	Вычислить агрегат для группы и вернуть его для каждой строки этой группы, сохраняя детализацию.	Свернуть (схлопнуть) группу строк в одну, показав только итоговый агрегат.
Количество строк в результате	Равно количеству строк в исходной таблице (или после WHERE).	Равно количеству уникальных групп, определенных в GROUP BY.
Когда использовать	«Для каждого товара показать его цену и среднюю цену в его категории».	«Показать среднюю цену по каждой категории».

1.3 Ограничения и частые ошибки

Оконные функции **нельзя** использовать в предложении **WHERE**.

WHERE обрабатывается **до того**, как вычисляются оконные функции.

СУБД сначала отбирает строки, и только потом применяет к ним оконные вычисления.

Листинг 8. Правильный подход – использованием CTE

```
WITH RankedData AS (
    SELECT
        product_name,
        price,
        -- Сначала вычисляем ранг в CTE
        RANK() OVER (PARTITION BY category_id ORDER BY price DESC) AS
        price_rank
    FROM products
)
-- А затем фильтруем по нему во внешнем запросе
SELECT * FROM RankedData WHERE price_rank <= 3;
```

Таким образом, чтобы отфильтровать результат по значению оконной функции (например, выбрать строки с рангом `price_rank <= 3`), необходимо сначала вычислить это значение, а потом применить фильтр с помощью подзапроса или обобщенного табличного выражения (CTE)

2. Категории оконных функций

2.1 Ранжирующие функции (ROW_NUMBER, RANK, DENSE_RANK)

Эти функции присваивают числовой ранг каждой строке внутри раздела в соответствии с порядком, заданным в **ORDER BY**.

Основное различие проявляется при обработке строк с одинаковыми значениями («дублей»).

Функция	Как работает с дублями	Пример (для значений 100, 90, 90, 80)	Когда использовать
ROW_NUMBER()	Присваивает уникальный номер каждой строке, независимо от совпадений	1, 2, 3, 4	Когда нужно гарантированно пронумеровать строки без повторов , например, для страничного вывода.
RANK()	Присваивает одинаковый ранг строкам-дублям, но пропускает следующий ранг .	1, 2, 2, 4	Классические рейтинги , где после двух человек на втором месте следующий будет на четвертом (<i>например, спорт</i>).
DENSE_RANK()	Присваивает одинаковый ранг строкам-дублям, но не пропускает следующий ранг .	1, 2, 2, 3	Идеально для задач «найти топ-3» , где нужно включить всех , кто делит призовое место, без пропусков.

ROWS — позиционно (считает строки)

- **Нарастающий итог (до текущей)**

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
(по умолчанию при наличии ORDER BY)

- **Последние N строк + текущая (скользящее окно)**

ROWS BETWEEN N PRECEDING AND CURRENT ROW

- **Окно «вперёд» (от текущей)**
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
- **Только текущая строка**
ROWS BETWEEN CURRENT ROW AND CURRENT ROW (или шорткат — ROWS CURRENT ROW)
- **Весь раздел**
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
- **Центрированное окно ($\pm N$ соседей)**
ROWS BETWEEN N PRECEDING AND N FOLLOWING

RANGE — по значению ключа сортировки

- **Нарастающий итог (до текущего значения)**
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- **Последние N единиц значения + текущее**
RANGE BETWEEN <expr> PRECEDING AND CURRENT ROW

Примеры:

- **по дате/времени:**
RANGE BETWEEN INTERVAL '90 day' PRECEDING AND CURRENT ROW
- **по числу:**
RANGE BETWEEN 100 PRECEDING AND CURRENT ROW
- **Окно «вперёд» (от текущего значения)**
RANGE BETWEEN CURRENT ROW AND <expr> FOLLOWING
Примеры: INTERVAL '7 day' FOLLOWING / 10 FOLLOWING
- **Только текущее значение (все «пиры» с тем же ключом)**
RANGE BETWEEN CURRENT ROW AND CURRENT ROW

- **Весь раздел**
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
- **Центрированное окно по значению ($\pm N$ единиц)**
RANGE BETWEEN <expr> PRECEDING AND <expr> FOLLOWING

GROUPS — по группам «пиров» (равные значения ORDER BY)

- **Нарастающий итог «по группам» (до текущей группы)**
GROUPS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- **Последние N групп + текущая**
GROUPS BETWEEN N PRECEDING AND CURRENT ROW
- **Окно «вперёд» по группам**
GROUPS BETWEEN CURRENT ROW AND N FOLLOWING
- **Только текущая группа пиров**
GROUPS BETWEEN CURRENT ROW AND CURRENT ROW
- **Весь раздел (все группы)**
GROUPS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
- **Центрированное окно по группам ($\pm N$ групп)**
GROUPS BETWEEN N PRECEDING AND N FOLLOWING

Памятка (общая для всех)

- Все рамки включают границы (inclusive).
- Без ORDER BY тонкая рамка теряет смысл — по сути будет весь раздел.
- Односторонние «шорткаты» допустимы (например, ROWS CURRENT ROW), но полная форма **всегда**:

[FRAME] BETWEEN <start> AND <end>,

где <start> = **UNBOUNDED PRECEDING** | **N PRECEDING** | **CURRENT ROW**,
<end> = **CURRENT ROW** | **N FOLLOWING** | **UNBOUNDED FOLLOWING**.

2.2 Агрегатные оконные функции (SUM, AVG, COUNT)

Стандартные агрегатные функции могут работать как оконные. В сочетании с **ORDER BY** они позволяют легко вычислять нарастающие (кумулятивные) итоги.

Как это работает: когда в **OVER()** указан **ORDER BY**, по умолчанию окно (фрейм) включает все строки от начала раздела до текущей строки.

SUM(...) **OVER (PARTITION BY ... ORDER BY ...)** просуммирует значения не по всему разделу, а только от его начала до текущей строки.

2.3 Функции смещения (LAG, LEAD)

Эти функции позволяют «заглянуть» в соседние строки, не используя сложные соединения таблицы с самой собой.

LAG(столбец, смещение, значение_по_умолчанию): возвращает значение из столбца строки, которая находится на смещение позиций раньше текущей.

LEAD(столбец, смещение, значение_по_умолчанию): возвращает значение из столбца строки, которая находится на смещение позиций позже текущей.

Ключевые моменты:

смещение: количество строк, на которое нужно «отступить». По умолчанию имеет значение «1».

значение_по_умолчанию: что вернуть, если «соседней» строки не существует (например, для первой строки нет предыдущей). Использование этого параметра помогает избежать NULL в результате.

3. Построение сводных таблиц (Pivoting)

Pivoting – это преобразование данных из «длинного» формата в «широкий», когда значения из одного столбца становятся заголовками новых столбцов.

3.1 Сравнение подходов

Критерий	Условная агрегация (SUM + CASE)	Функция crosstab
Совместимость	Стандарт ANSI SQL. Работает в большинстве СУБД.	Специфична для PostgreSQL. Требуется установки расширения 'tablefunc'.
Синтаксис	Более интуитивный и простой, если количество итоговых столбцов известно заранее и невелико.	Более сложный, требует описания структуры результата и подготовки специального запроса-источника.
Гибкость	Статичный. Названия и количество столбцов должны быть жестко прописаны в запросе.	Динамический. Может создавать столбцы на основе данных, которые есть в таблице на момент выполнения.
Потенциальные проблемы	Запрос становится громоздким при большом количестве итоговых столбцов.	Сложная отладка. Ошибки в запросе-источнике или в описании типов данных могут приводить к неочевидным сообщениям об ошибках.

3.2 Ключевые моменты для crosstab

1. Перед первым использованием выполните команду **CREATE EXTENSION IF NOT EXISTS tablefunc;**
2. **Запрос-источник:** crosstab принимает на вход SQL-запрос в виде строки, который должен возвращать ровно три столбца:
 - **row_name:** идентификатор, который станет строкой в результате.
 - **category:** категория, которая станет столбцом.

- **value:** значение, которое окажется в ячейке.

3. **Обязательная сортировка:** результат запроса-источника обязательно должен быть отсортирован по первому, а затем по второму столбцу (**ORDER BY 1, 2**).

Нарушение этого правила – самая частая причина ошибок.

4. **Описание результата:** после вызова `crosstab` необходимо явно определить структуру итоговой таблицы, включая названия и типы данных всех столбцов.