MySQL

АГРЕГИРОВАНИЕ, ГРУППИРОВКА ДАННЫХ И ОКОННЫЕ ФУНКЦИИ

СОДЕРЖАНИЕ

АГРЕГИРОВАНИЕ И ГРУППИРОВКА ДАННЫХ

ВВЕДЕНИЕ

- Мы уже подробно разбирали выборку из таблиц, когда мы могли получить не целиком все данные, а некий набор информации в ограниченном виде. Например, в одном случае речь могла идти только о значениях определенных столбцов, в другом случае мы накладывали на выборку определенные условия и получали только часть строк таблицы. Бывали даже случаи, когда мы объединяли значения нескольких столбцов в один столбец при помощи функции **CONCAT**. Заметим, что во всех этих случаях мы работали с данными, которые были непосредственно занесены в таблицу. Однако часто нам надо получить не первичные данные, а более обобщенную информацию. Например, какое-либо среднее значение по всей таблице (средняя зарплата сотрудников) или даже среднее значение в контексте неких групп строк (средняя зарплата только среди женщин определенной национальности). Короче говоря, данные в таблице могут сложиться в некие закономерности или статистические множества, которые мы хотели бы быстро получить.

- Вышеупомянутый процесс выборки также известен как "**агрегирование данных**". Вот его полное определение - "агрегирование данных - тип процесса извлечения данных и информации, при котором данные ищутся, собираются и представляются в обобщенном формате". Если упростить - процесс получения одного или несколько обобщенных значений из больших массивов данных.

- Осуществить агрегирование можно разными способами - есть вариант при помощи какого-либо языка программирования выбрать все данные из таблицы, а уже потом посчитать их, используя соответствующие математические приемы. Однако это может оказать очень затратно по времени и потреблению памяти (это станет особенно очевидно если в таблице будут миллионы строк). Поэтому в MySQL есть внутренние инструменты для агрегирования. В первую очередь это функции **COUNT**, **AVG**, **MAX**, **MIN**, **SUM** и **GROUP_CONCAT**. Также нам предоставляется возможность фильтровать уже агрегированные данные при помощи оператора **HAVING**. Кроме того, мы можем создавать разные группы данных и уже в них производить некие обобщающие выборки - это осуществляется при помощи оператора **GROUP BY**.

- Перед тем, как перейти непосредственно к конструкциям агрегирования, создадим три необходимые для нас таблицы: tasks, workers и workers_tasks (теоретически, мы могли бы обойтись одной таблицей, но для лучшего понимания поработаем сразу в нескольких одновременно). После создания они должны выглядеть следующим образом:

Таблица **tasks**:

id	**title**
1	Accounting Automatization
2	AI Algorithms Design
3	Site Development

Таблица **workers**:

id	**name**	**department**	**nationality**	**sex**	**salary**
1	Eleonore Bergmann	Head Office	German	woman	7800.00
2	Ennio Salieri	Development	Italian	man	5200.00
3	NULL	Head Office	American	woman	5100.00
4	John Smith	Sales	Canadian	man	2300.00
5	Helene Kempf	Development	German	woman	900.00
6	Anna Johnson	Sales	American	woman	1100.00
7	Tommy Angelo	Sales	Italian	man	3700.00
8	Rosa Hawkins	Development	American	woman	2300.00
9	Frank Colletti	Head Office	Italian	man	4874.00
10	Johan Hofmann	Sales	German	man	2700.00

Таблица **workers_tasks:**

worker_id	**task_id**
1	1
2	2
3	2
4	1
5	3
6	1
7	1
8	3
9	2
10	1
1	2
1	3

ОСНОВЫ РАБОТЫ С АГРЕГАТНЫМИ ФУНКЦИЯМИ

- Самое простое задание, которое можно решить с помощью агрегатных функций - посчитать общее количество строк в таблице. Для этого подойдет функция **COUNT**, которой надо передать звездочку. Посчитаем количество всех работников:

```
SELECT COUNT(*) FROM workers;
```

COUNT(*)
10

- Теоретически для подсчета количества строк в таблице, функции **COUNT** можно передать название одного любого столбца таблицы, но в этом случае есть опасность, что некоторые строки будут пропущены и общее количество будет ошибочным. Попробуем посчитать строки при помощи столбца name:

```
SELECT COUNT(name) FROM workers;
```

COUNT(name)
9

Это может показаться странным, но мы получим в ответе не 10, а 9. На самом деле ничего странного нет - мы помним, что значение **NULL** в SQL означает отсутствие всякого значения. В нашей таблице workers работник с id равным 3 имеет имя равное **NULL**. Если агрегатная функция использует для выборки не звездочку, а столбец, и для какой-то строки этот столбец будет равен **NULL**, то информация об этой строке не попадет результат. Поэтому при общих подсчетах лучше использовать звездочку, а название столбца применять там, где мы хотим подсчитать строки, у которых этот столбец не равен **NULL**.

- Также агрегатные функции умеют работать с уникальными значениями. Например, мы хотим посчитать количество отделов, в которых задействованы наши работники. Попробуем передать в **COUNT** название столбца department:

```
SELECT COUNT(department) FROM workers;
```

COUNT(department)
10

Мы получим 10, но это не совсем то, что нам нужно. Дело в том, что у некоторых работников отделы одинаковые - следовательно в подсчет некоторые отделы попали несколько раз. Чтобы разрешить это противоречие просто подставим уже знакомое нам ключевое слово **DISTINCT** перед название столбца и получим точный результат (3 отдела):

```
SELECT COUNT(DISTINCT department) FROM workers;
```

COUNT(DISTINCT department)
3

- Агрегатные функции подчиняются фильтрации (конструкция **WHERE**) - как и стандартные значения столбцов таблицы. Однако

запомним простое правило - сначала происходит фильтрация, а уже потом по оставшимся после фильтрации значениям столбцов происходит вычисление результата функции. Учитывая это, попробуем посчитать то количество работников, которые являются женщинами (кстати, агрегатному столбцу можно давать псевдоним, что мы в этом случае и сделаем):

```
SELECT COUNT(*) AS quantity FROM workers WHERE sex = 'woman';
```

quantity
5

- Как уже было сказано, в MySQL кроме **COUNT** существуют и другие агрегатные функции, которые могут быть полезны при работе с таблицами. Для начала рассмотрим функцию **AVG**, которая возвращает среднее арифметическое для какого-либо числового столбца. Если передать этой функции столбец с другим типом, например текстовым, то ошибки не будет, но результат всегда будет равным нулю(0). Оператор звездочка (*) не поддерживается - если попытаться передать ее, произойдет ошибка. Попробуем использовать **AVG**, чтобы посчитать среднюю зарплату всех работников:

```
SELECT AVG(salary) FROM workers;
```

AVG(salary)
3597.400000

Теперь попробуем получить среднюю зарплату только у отдела разработки (Development):

```
SELECT AVG(salary) FROM workers
WHERE department = 'Development';
```

AVG(salary)
2800.000000

- Агрегатные функции **MIN** и **MAX** вычисляют соответственно минимальное и максимальное значение для тех столбцов

числового типа, которые переданы в них. Как и в случае с **AVG**, передача звездочки вызовет ошибку, но если передать текстовый столбец, то будет получены лексикографически первое (функция **MIN**) и лексикографически последнее (функция **MAX**) значение столбца. Кстати, несколько агрегирующих функции можно использовать одновременно в одном запросе. Например, найдем минимальную и максимальную зарплату работников в одном запросе:

```
SELECT
    MIN(salary) AS min_salary,
    MAX(salary) AS max_salary
FROM workers;
```

min_salary	max_salary
900.00	7800.00

Далее попытаемся найти минимальную и максимальную зарплату только среди мужчин:

```
SELECT
    MIN(salary) AS man_min_salary,
    MAX(salary) AS man_max_salary
FROM workers
WHERE sex = 'man';
```

man_min_salary	man_max_salary
2300.00	5200.00

- Еще одна функция - **SUM** - подсчитывает общую сумму значений того числового столбца, который мы передали в эту функцию. Логика для столбцов других типов такая же как и у **AVG** - ошибка для звездочки, 0 для текстовых столбцов. Применим **SUM** для получения общей зарплаты для всех работников:

```
SELECT SUM(salary) FROM workers;
```

SUM(salary)
35974.00

Теперь узнаем общую сумму зарплаты людей, работающих в головном офисе (Head Office):

```
SELECT SUM(salary) FROM workers
WHERE department = 'Head Office';
```

SUM(salary)
17774.00

- Функция **GROUP_CONCAT** буквально трактует значение слова “обобщение” и просто объединяет все значения переданного ей столбца в строку, используя запятую в качестве разделителя. Получим имена всех работников с помощью этой любопытной конструкции:

```
SELECT GROUP_CONCAT(name) AS names FROM workers;
```

names
Eleonore Bergmann,Ennio Salieri,John Smith,Helene Kempf,Anna Johnson,Tommy Angelo,Rosa Hawkins,Frank Colletti,Johan Hofmann

Однако запятая не является обязательным разделителем. При помощи ключевого слова **SEPARATOR** мы можем задать свой разделитель в виде текста. Например, получим имена работников отдела продаж, разделенные амперсандом (&):

```
SELECT GROUP_CONCAT(name SEPARATOR ' & ') AS sales_names
FROM workers WHERE department = 'Sales';
```

sales_names
John Smith & Anna Johnson & Tommy Angelo & Johan Hofmann

Напоследок заметим, что **GROUP_CONCAT** поддерживает сортировку значений внутри себя. Например, мы хотим, чтобы в предыдущем запросе имена сотрудников были бы отсортированы по их зарплате - от большей к меньшей. Для этого вы выполним сортировку **ORDER BY** внутри самой функции:

```
SELECT
    GROUP_CONCAT(name ORDER BY salary ASC SEPARATOR ' & ') AS
sorted_sales_names
FROM workers WHERE department = 'Sales';
```

sorted_sales_names
Anna Johnson & John Smith & Johan Hofmann & Tommy Angelo

- Интересно заметить, что результаты функций могут взаимодействовать в рамках одного запроса. Например, попробуем узнать, какой процент от общей суммы зарплат (**SUM**), занимает зарплата работника с максимальной зарплатой (**MAX**):

```
SELECT
    MAX(salary) / SUM(salary) * 100 AS max_salary_percent
FROM workers;
```

max_salary_percent
21.682326

- Использование агрегатных функций возможно и в более сложных запросах с участием множества таблиц. Например получим общую сумму зарплат и строку с именами тех работников (таблица workers), которые работают над заданием (таблицы tasks и workers_tasks) по созданию сайта (Site Development):

```
SELECT SUM(w.salary), GROUP_CONCAT(w.name)
FROM tasks t
JOIN workers_tasks wt ON t.id = wt.task_id
JOIN workers w ON wt.worker_id = w.id
WHERE t.title = 'Site Development';
```

SUM(w.salary)	GROUP_CONCAT(w.name)
11000.00	Eleonore Bergmann,Helene Kempf,Rosa Hawkins

- Предыдущий является корректным, он будет работать при любых значениях столбца t.title. Однако может показаться, что корректность запроса будет сохраняться даже в том случае, если мы захотим узнать сумму зарплат работников, которые

заняты на каких-либо двух (или более) заданиях. Попробуем это реализовать с помощью запроса, похожего на предыдущий, для проектов Site Development и AI Algorithms Design:

```
SELECT SUM(w.salary), GROUP_CONCAT(w.name)
FROM tasks t
JOIN workers_tasks wt ON t.id = wt.task_id
JOIN workers w ON wt.worker_id = w.id
WHERE t.title IN('Site Development', 'AI Algorithms Design');
```

SUM(w.salary)	GROUP_CONCAT(w.name)
33974.00	Eleonore Bergmann,Ennio Salieri,Frank Colletti,Eleonore Bergmann,Helene Kempf,Rosa Hawkins

Результат не тот, какой нам был нужен. Дело в том, что Eleonore Bergmann участвует сразу в двух заданиях одновременно (это можно заметить уже по результату выборки **GROUP_CONCAT** - ее имя появляется там два раза), и ее зарплата была посчитана два раза. Что же делать в таких случаях? Конечно, можно поставить **DISTINCT** на w.name и w.salary, но это тоже неправильно, т.к. у двух или более людей могут быть одинаковые имена и зарплаты - в этом случае их зарплаты не будут суммироваться. На самом деле решение очень простое - мы сначала выберем уникальные комбинации идентификаторов, имен и зарплат (даже если зарплаты и имена будут одинаковые, идентификаторы их владельцев все равно будут разные), а затем уже применим группировку на получившийся результат. Мы должны помнить, что запрос всегда возвращает таблицу, а на таблицу всегда можно сделать другой запрос и так много раз (главное не забыть дать каждому промежуточному результату свой псевдоним):

```
SELECT SUM(r.salary), GROUP_CONCAT(r.name)
FROM
(
    SELECT DISTINCT w.id, w.salary, w.name
    FROM tasks t
    JOIN workers_tasks wt ON t.id = wt.task_id
    JOIN workers w ON wt.worker_id = w.id
    WHERE t.title IN('Site Development', 'AI Algorithms
Design')
) r;
```

SUM(w.salary)	GROUP_CONCAT(w.name)
26174.00	Eleonore Bergmann,Ennio Salieri,Frank Colletti,Helene Kempf,Rosa Hawkins

ГРУППИРОВКА ДАННЫХ

- Внимательный читатель/зритель мог заметить, что в предыдущей подглаве мы запрашивали в рамках **SELECT** только результаты агрегатных функций, но ни разу не запросили какое-либо значение конкретного столбца. Например, при подсчете работников мы не запрашивали их зарплату следующим образом:

```
SELECT COUNT(*), salary FROM workers;
```

Если мы выполним такой запрос, не меняя настроек MySQL по умолчанию, то получим ошибку. Если немного подумать, то такое поведение абсолютно адекватно, ведь мы не можем узнать, зарплату какого работника мы пытаемся получить в данном контексте - ведь агрегатные функции преобразовывают много значений в одно, т.е. непонятно к кому отнести эту зарплату. Кстати, если мы очень хотим, то получить хотя бы какое-то значение salary все же можно. Для этого надо зайти в файл настроек my.ini, убрать из параметра sql-mode значение **ONLY_FULL_GROUP_BY**, сохранить **my.ini** и перезапустить MySQL сервер. В этом случае MySQL скорее всего вернет зарплату первого попавшего под подсчет сотрудника (в нашем случае это будет Eleonore Bergmann), но это не гарантируется со стопроцентной уверенностью даже при применении сортировки:

COUNT(*)	salary
10	7800.00

Однако запомним раз и навсегда - именно так запрашивать данные не следует. Не следует совмещать результаты агрегатных функций и значения столбцов в простых запросах без дополнительных условий. Во-первых это не имеет особого смысла (а если быть точным - не имеет вообще никакого смысла), во-вторых, это возможно только при манипуляциях с настройками, и, в-третьих, нам не гарантируется, что значение запрашиваемого столбца будет всегда одним и тем же.

- Однако существуют запросы определенного типа, в которых присутствие дополнительных столбцов не только не порицается, а крайне приветствуется. Давайте представим, что нам надо узнать среднюю зарплату для каждого пола, общее количество женщин для каждой национальности или количество проектов для каждого работника. В каждом из этих запросов кроме агрегатного значения должен может присутствовать еще один столбец (пол, национальность или идентификатор работника), который и характеризуются агрегатным значением. По сути мы мы выделяем разные группы, а уже потом для каждой из этих групп считаем обобщающее агрегатное значение. Например, группируем таблицу по столбцу национальность, а потом считаем сумму зарплат для каждой группы - это значит, что у итальянцев будет своя одна сумма, состоящая только из их зарплат, у немцев - вторая, у американцев - третья и т.д.

- Как же дать понять нашей СУБД, что те столбцы, которые не являются агрегатными и которые мы укажем в запросе, предназначены для групировки? Ведь если ничего не делать, то в лучшем случае нам вернется какое-то непонятное значение, а в худшем - произойдет ошибка. Для этого в MySQL существует конструкция **GROUP BY**, после которой нужно перечислить те самые столбцы для группировки. Например, реализуем наш запрос для подсчета общей суммы зарплат в рамках каждой национальности - после **GROUP BY** укажем столбец nationality, а также запросим значение столбца nationality после **SELECT**:

```
SELECT nationality, SUM(salary)
FROM workers
GROUP BY nationality;
```

nationality	SUM(salary)
German	11400.00
Italian	13774.00
American	8500.00
Canadian	2300.00

- Далее попробуем применить группировку по нескольким столбцам - узнаем какая средняя зарплата у тех людей, которые имеют один и тот же пол и относятся к одному и тому же отделу. Не будет сюрпризом, что для этого нам надо перечислить через

запятую названия двух столбцов (sex и department) после конструкции **GROUP BY:**

```
SELECT sex, department, SUM(salary) FROM workers
GROUP BY department, sex;
```

Заметим, что в группировке (**GROUP BY**) и в выборке (**SELECT**) мы указали названия столбцов в разном порядке - в данном случае это не имеет особого значения - можем определять такой порядок, который нам удобен.

sex	department	SUM(salary)
woman	Head Office	12900.00
man	Development	5200.00
man	Sales	8700.00
woman	Development	3200.00
woman	Sales	1100.00
man	Head Office	4874.00

- Также попробуем применить группировку к данным, которые могут находиться в разных таблицах - получим количество заданий (tasks) для каждого работника (workers):

```
SELECT w.id, w.name, COUNT(t.id) AS qty
FROM workers w
JOIN workers_tasks wt ON w.id = wt.worker_id
JOIN tasks t ON wt.task_id = t.id
GROUP BY w.id, w.name;
```

Обратим внимание, что группировка проходит не по полю name, а по 2 столбцам - id и name. Ведь имена у работников могут оказаться одинаковыми, а комбинация идентификатора и имени в таблице workers будут уникальными всегда. Может показаться, что для группировки нам бы хватило только одного столбца id таблицы workers, т.к. он является первичным ключом и уже сам по себе будет уникальным. Это действительно так, но нам кроме ключа еще надо получить имя работника, а если мы его не укажем в группировке, то произойдет ошибка. Еще одна особенность - одно из полей name имело значение NULL, но все

равно строка с ним была включена в результат. Это значит, что группировка (в отличие от агрегирования) учитывает NULL.

id	name	qty
1	Eleonore Bergmann	3
4	John Smith	1
6	Anna Johnson	1
7	Tommy Angelo	1
10	Johan Hofmann	1
2	Ennio Salieri	1
3	NULL	1
9	Frank Colletti	1
5	Helene Kempf	1
8	Rosa Hawkins	1

- Из всех предыдущих запросов можно сделать следующий вывод - если в запросе присутствует хотя бы одна агрегатная функция, то в выборке (после **SELECT** через запятую) может присутствовать только результат этой функции (или результаты этих нескольких функций), а также только значения тех столбцов, которое участвуют в группировке (перечислены через запятую после **GROUP BY**), если такая группировка осуществляется. Больше ничего в выборке указывать нельзя (разве что некоторые объединения уже разрешенных столбцов или искусственные столбцы, но это уже экзотика).

- Мы уже видели, что агрегатные запросы могут работать вместе с фильтрацией. Эта возможность доступна и для запросов, в которых применяется группировка. Допустим мы хотим получить среднюю зарплату для работников разных полов, но только в том случае, если зарплата будет более 2000 евро. Запрос будет выглядеть следующим образом:

```
SELECT sex, AVG(salary) FROM workers
WHERE salary > 2000
GROUP BY sex;
```

Запрос выполнится нормально, но в таких случаях мы всегда должны помнить - сначала отработает фильтрация (выполняться все условия, которые перечислены в **WHERE**), а уже затем произойдет группировка и агрегирование. Далее результат нашего запроса:

sex	AVG(salary)
woman	5066.666667
man	3754.800000

- Еще упомянем, что группировка отлично работает в том случае, когда у нас в запросе присутствуют сразу несколько функций агрегирования. Для примера попробуем выбрать самую большую (**MAX**) и самую маленькую зарплату (**MIN**) в пределах каждого отдела, а также одновременно получить в виде строки имена всех работников этого отдела (**GROUP_CONCAT**):

```
SELECT
    department,
    MIN(salary) AS min_salary,
    MAX(salary) AS max_salary,
    GROUP_CONCAT(name) AS names
FROM workers
GROUP BY department;
```

department	min_salary	max_salary	names
Development	900.00	5200.00	Ennio Salieri,Helene Kempf,Rosa Hawkins
Head Office	4874.00	7800.00	Eleonore Bergmann,Frank Colletti
Sales	1100.00	3700.00	John Smith,Anna Johnson,Tommy Angelo,Johan Hofmann

- Как мы могли заметить, при группировке у нас в результирующей таблице может появиться уже несколько строк, а не одна, как это было в случае простого агрегирования. Это значит, что появилась возможность применять сортировку и накладывание лимита на количество возвращаемых строк (такая возможность есть и при простой группировке, просто это не будет оказывать никакого эффекта, ведь возвращается одна

строка). Сразу определим порядок операций - сначала происходит группировка (**GROUP BY**) и агрегирование, затем сортировка (**ORDER BY**), а уже в самом конце ограничение количества строк (**LIMIT**). Для примера получим среднюю зарплату для каждого отдела, затем отсортируем по названию отдела в алфавитном порядке, а затем возьмем две первых строки (отделов всего 3, поэтому один должен быть отброшен):

```
SELECT department, AVG(salary) AS avg_salary FROM workers
GROUP BY department
ORDER BY department ASC
LIMIT 2;
```

department	avg_salary
Development	2800.000000
Head Office	5924.666667

- Сортировка по тому полю, которое участвует в группировке ничем не отличается от стандартной сортировки. А что делать, если нам надо отсортировать непосредственно по значению, которое будет получено в результате агрегирования, т.е. расположить отделы в порядке возрастания их средней зарплаты? Есть много способов, но приведем самые простые:

```
-- 1-й способ - с помощью самой агрегатной функции
SELECT department, AVG(salary) AS avg_salary FROM workers
GROUP BY department ORDER BY AVG(salary) ASC LIMIT 2;

-- 2-й способ - по псевдониму результата агрегатной функции
SELECT department, AVG(salary) AS avg_salary FROM workers
GROUP BY department ORDER BY avg_salary ASC LIMIT 2;

-- 3-й способ - по номеру необходимого нам столбца в выборке
SELECT department, AVG(salary) AS avg_salary FROM workers
GROUP BY department ORDER BY 2 ASC LIMIT 2;
```

department	avg_salary
Sales	2450.000000
Development	2800.000000

ПОЛУЧЕНИЕ ИТОГОВЫХ ДАННЫХ ПРИ ГРУППИРОВКЕ

- Группировка - замечательный механизма, который в умелых руках может принести много пользы. Однако ему не хватает одной маленькой особенности - получая обобщенные данные по конкретным группам столбцов, мы не можем только при помощи **GROUP BY** одновременно получить данные по всем столбцам сразу или по какой-то части столбцов, входящих в группировку. К счастью, MySQL предоставляет возможность это сделать с помощью специальной конструкции **WITH ROLLUP**, которая предоставляет итоговые данные абсолютно для всех возможных комбинаций столбцов группировки. Для примера попробуем выбрать среднюю зарплату для всех национальностей всех отделов:

```
SELECT department, nationality, AVG(salary) AS avg_salary
FROM workers
GROUP BY department, nationality WITH ROLLUP;
```

department	nationality	avg_salary
Development	American	2300.000000
Development	German	900.000000
Development	Italian	5200.000000
Development	NULL	2800.000000
Head Office	American	5100.000000
Head Office	German	7800.000000
Head Office	Italian	4874.000000
Head Office	NULL	5924.666667
Sales	American	1100.000000
Sales	Canadian	2300.000000
Sales	German	2700.000000
Sales	Italian	3700.000000
Sales	NULL	2450.000000
NULL	NULL	3597.400000

Можно увидеть, что конструкция **WITH ROLLUP** вставила строки, содержащие итоговые данные и по всей таблице сразу, и по каждому отделу без учета национальностей, а также то, что мы просили изначально - среднюю зарплату в контексте каждой национальности каждого отдела. Те же значения столбцов, которые на тот момент не учитывались, были помечены значением NULL.

ФИЛЬТРАЦИЯ АГРЕГИРОВАННЫХ ДАННЫХ

- Мы уже неоднократно говорили о том, что агрегирование и группировка происходит только после основной фильтрации (применение условий конструкции **WHERE**). У внимательного слушателя может возникнуть неожиданный вопрос - а что, если нам надо отфильтровать данные после их агрегирования? Например, попробовать выбрать только те национальности работников, которые имеют среднюю зарплату выше 3000 евро. Среднюю зарплату мы получаем после агрегирования - что же нам делать? MySQL позволяет нам решить эту проблему - на этот раз при помощи конструкции **HAVING**, которая отработает и как после **WHERE**, так и после **GROUP BY**.

```
-- 1-й способ - фильтрация по функции агрегирования
SELECT nationality, AVG(salary) AS avg_salary
FROM workers
GROUP BY nationality
HAVING AVG(salary) > 3000;

-- 2-й способ - фильтрация по псевдониму
SELECT nationality, AVG(salary) AS avg_salary
FROM workers
GROUP BY nationality
HAVING avg_salary > 3000;
```

nationality	avg_salary
German	3800.000000
Italian	4591.333333

- **HAVING** поддерживает не только одно условие, а сколько угодно условий - например, повторим предыдущий запрос, но одновременно добавим взятие максимальной зарплаты для каждой национальности, а затем в агрегированной фильтрации укажем, что максимальная зарплата должна быть больше семи тысяч:

```
SELECT
    nationality,
    AVG(salary) AS avg_salary,
    MAX(salary) AS max_salary
FROM workers
GROUP BY nationality
HAVING avg_salary > 3000 AND max_salary > 7000;
```

nationality	avg_salary	max_salary
German	3800.000000	7800.00

- **HAVING** можно применять и для агрегированных данных, которые не участвуют в выборке или группировке. Например, возьмем среднюю зарплату (**AVG(salary)**), сгруппированную по национальностям, но в условии **HAVING** применим проверку для для максимальной зарплаты (**MAX(salary)**), которая должна быть меньше трех тысяч (т.е. выбирается средняя зарплата, а проверка рассчитывается по максимальной):

```
SELECT nationality, AVG(salary) AS avg_salary
FROM workers
GROUP BY nationality
HAVING MAX(salary) < 3000;
```

nationality	avg_salary
Canadian	2300.000000

Такой подход будет работать даже в том случае, если мы применим агрегированную фильтрацию по тем столбцам, которые вообще нигде не упоминаются (повторим предыдущий запрос, но не будем вообще указывать salary после **SELECT**):

```
SELECT nationality
FROM workers
GROUP BY nationality
HAVING MAX(salary) < 3000;
```

nationality
Canadian

- Применение **HAVING** не следует ограничивать только агрегатными функциями - также с помощью этой директивы можно применять фильтрацию по простым столбцам, участвующим в выборке. Например, посмотрим, какая средняя зарплата у итальянцев и американцев (это можно сделать с помощью **WHERE**, но для примера попробуем применить именно **HAVING**):

```
SELECT nationality, AVG(salary) AS avg_salary
FROM workers
GROUP BY nationality
HAVING nationality IN ('Italian', 'American');
```

nationality	avg_salary
Italian	4591.333333
American	2833.333333

- В завершение этой подглавы укажем одно интересное противоречие **HAVING**. Мы уже знаем, что эта директива может работать с любыми агрегированными значениями (даже для тех полей, которые в этом запросе больше нигде не упоминаются) или с теми простыми столбцами, которые упоминаются в выборке. Попробуем повторить предыдущий запрос, но попытаемся отфильтровать его при помощи **HAVING** только для департамента продаж (Sales):

```
SELECT nationality, AVG(salary) AS avg_salary
FROM workers
GROUP BY nationality
HAVING department = 'Sales';
```

В результате мы получим ошибку и более ничего. **HAVING** видит только те столбцы, значения которых агрегированы, а также те простые столбцы, которые участвуют в выборке. Остальные столбцы и их значения для этой конструкции просто не существуют.

ОКОННЫЕ ФУНКЦИИ

ВВЕДЕНИЕ

- Агрегирование и группировка очень хорошо помогают при получении статистической и прочей обобщающей информации, но у этих конструкций есть один небольшой недостаток. В прошлой подглаве мы могли получать информацию по отделам, в которых работают люди, среднюю зарплату национальностей, распределение полов и т.д. Однако очень часто необходима информация немного другого качества - какая средняя зарплата в том отделе, в котором работает **каждый** конкретный человек, сколько заданий выполняет представители национальности **каждого** конкретного человека, какая максимальная зарплата у того пола, которому принадлежит **каждый** конкретный человек. Это значит, что иногда нам не надо сокращать количество строк, чтобы получить обобщающую информацию - часто надо получить обобщающую информацию для **каждой** строки.

- Теоретически можно попытаться получить обобщающую информация по строкам без всяких нововведений. Например, при помощи какого-либо языка программирования получить всех работников, а затем для каждого последовательно выбрать всю информацию. Однако это очень затратно по времени и памяти - такая программа будет работать очень неэффективно. Есть вариант использовать конструкцию **WITH ROLLUP**, но с одной стороны она вернет слишком большую таблицу со всеми комбинациями результатов, а с другой - каждый результат агрегатного столбца будет подчиняться одной и той же группировке, а часто нам нужна разная группировка.

- Решение нашей проблемы крайне простое - надо использовать т.н. оконные функции. По своей сути это уже знакомые нам функции (и еще некоторые другие, которые мы обсудим позже) - **COUNT**, **SUM** и т.д., после которых следует конструкция **OVER()**, задающая “окно”, осуществляющее агрегацию для каждой строки. Внутри конструкции **OVER** может располагаться группировка по столбцам - на это раз она осуществляется с помощью конструкции **PARTITION BY**, например, вот так: **COUNT(*) OVER (PARTITION BY department)**. Также внутри **OVER** можно осуществлять сортировку: **OVER (PARTITION BY department ORDER BY salary)**. Позднее мы обсудим, зачем такая сортировка нужна и где ее можно (и даже обязательно) задействовать.

- Попытаемся применить новые знания на практике - получим среднюю зарплату для отдела в котором работает каждый человек и одновременно получим наивысшую зарплату, которую получает представитель пола каждого человека, а также одновременно общее количество работников:

```
SELECT name,
AVG(salary) OVER(PARTITION BY department) AS avg_dep_sal,
MAX(salary) OVER(PARTITION BY sex) AS max_sex_sal,
COUNT(*) OVER() AS total_qty
FROM workers;
```

name	avg_dep_sal	max_sex_sal	total_qty
Ennio Salieri	2800.000000	5200.00	10
Frank Colletti	5924.666667	5200.00	10
John Smith	2450.000000	5200.00	10
Tommy Angelo	2450.000000	5200.00	10
Johan Hofmann	2450.000000	5200.00	10
Helene Kempf	2800.000000	7800.00	10
Rosa Hawkins	2800.000000	7800.00	10
Eleonore Bergmann	5924.666667	7800.00	10
NULL	5924.666667	7800.00	10
Anna Johnson	2450.000000	7800.00	10

- Отметим, что часто конструкции **OVER** могут повторяться для нескольких столбцов (если мы хотим получить среднюю зарплату в для департамента, а потом получить наивысшую зарплату для департамента). В таких случаях можно выносить повторяющиеся конструкции в конец запроса с помощью ключевого слова **WINDOW**, а потом обращаться к ним по псевдониму:

```
SELECT name,
AVG(salary) OVER by_dep AS avg_dep_sal,
MAX(salary) OVER by_dep AS max_dep_sal
FROM workers
WINDOW by_dep as (PARTITION BY department);
```

name	avg_dep_sal	max_dep_sal
Ennio Salieri	2800.000000	5200.00
Helene Kempf	2800.000000	5200.00
Rosa Hawkins	2800.000000	5200.00
Eleonore Bergmann	5924.666667	7800.00
NULL	5924.666667	7800.00
Frank Colletti	5924.666667	7800.00
John Smith	2450.000000	3700.00
Anna Johnson	2450.000000	3700.00
Tommy Angelo	2450.000000	3700.00
Johan Hofmann	2450.000000	3700.00

- Важный момент - обычное агрегирование/группировку и применение оконных функция можно применять в одном и том же запросе, однако, во-первых, в данном случае оконные функции могут обрабатывать только те столбцы, которые перечислены в конструкции **GROUP BY**, и, во-вторых, оконные функции всегда будут срабатывать уже после того, как была произведена группировка. Это означает что совмещение обычных и оконных функций в большинстве случаев не приносит пользы.

- Всего есть четыре класса оконных функций:

1. **Агрегатные функции**
2. **Ранжирующие функции**
3. **Функции смещения**
4. Аналитические функции

Мы рассмотрим только первые три класса, т.к. аналитические функции, во-первых, требуют специальной математической подготовки (интегралы, процентили, постоянное распределение и т.д.), а, во-вторых, используются реже, чем другие. Перед рассмотрением классов создадим новую таблицу - devices, где каждое устройство будет иметь идентификатор, название, тип, количество, цену и год производства. Это необходимо, т.к. предыдущие таблицы не позволяют должным образом раскрыть весь возможный функционал оконных функций.

Таблица **devices:**

id	**name**	**type**	**quantity**	**price**	**c_year**
1	Apple Mac Mini M2	Computer	3	969	2019
2	Microsoft Surface	Tablet	2	455	2021
3	Samsung Odyssey	Monitor	4	227	2022
4	Lenovo Legion T7	Computer	2	815	2019
5	Nokia T20	Tablet	3	408	2023
6	Philips Envia 34M	Monitor	5	327	2022
7	Huawei MatePad	Tablet	2	473	2021
8	Dell Precision 3660	Computer	1	878	2022
9	Acer Nitro 50	Computer	7	928	2021
10	LG UltraWide 49W	Monitor	4	289	2019

АГРЕГАТНЫЕ ОКОННЫЕ ФУНКЦИИ

- Попробуем применить все известные нам агрегатные агрегатные функции - **SUM, AVG, MIN, MAX, COUNT** в оконном формате с разной группировкой. Это значит, что в каждой строке будет:

  1. Название
  2. Сумма количеств устройств данного типа (**SUM**)
  3. Средняя цена всех устройств в этом году (**AVG**)
  4. Минимальная цена устройства такого типа (**MIN**)
  5. Максимальная цена устройства в этом году (**MAX**)
  6. Общее количество всех устройств (**COUNT**)

  Кроме того, для разнообразия сделаем группировку для года через псевдоним, а остальные - прямо рядом с группировкой:

```
SELECT name,
SUM(quantity) OVER(PARTITION BY type) AS sum_type_qty,
AVG(price) OVER by_year AS avg_year_price,
MIN(price) OVER(PARTITION BY type) AS min_type_price,
MAX(price) OVER by_year AS max_year_price,
COUNT(*) OVER() AS total_device_qty
FROM devices
WINDOW by_year AS (PARTITION BY c_year);
```

name	sum_type_qty	avg_year_price	min_type_price	max_year_price	total_device_qty
Apple Mac Mini M2	13	691.000000	815.00	969.00	10
Lenovo Legion T7	13	691.000000	815.00	969.00	10
Acer Nitro 50	13	618.666667	815.00	928.00	10
Dell Precision 3660	13	477.333333	815.00	878.00	10
LG UltraWide 49W	13	691.000000	227.00	969.00	10
Samsung Odyssey	13	477.333333	227.00	878.00	10
Philips Envia 34M	13	477.333333	227.00	878.00	10
Microsoft Surface	7	618.666667	408.00	928.00	10
Huawei MatePad	7	618.666667	408.00	928.00	10
Nokia T20	7	408.000000	408.00	408.00	10

- Агрегатные оконные функции отлично работают с фильтрацией (**WHERE**), сортировкой (**ORDER BY**) и ограничением по количеству (**LIMIT**). Однако надо помнить, что сначала сработает фильтрация, потом будет выполнена агрегирование, а лишь затем сортировка и ограничение по количеству. Приведем пример, когда мы хотим получить только новые товары (год создания выше или равен 2022), отсортируем по средней цене в контексте типа, а потом возьмем две строки после первой:

```
SELECT name,
AVG(price) OVER (PARTITION BY type) AS avg_type_price
FROM devices
WHERE c_year >= 2022
ORDER BY avg_type_price LIMIT 1,2;
```

name	avg_type_price
Philips Envia 34M	277.000000
Nokia T20	408.000000

- Единственный функционал, которые в рамках оконных функций недоступен (на начало 2023 года) - применение слова **DISTINCT** непосредственно в функциях. Следующий запрос не будет выполнен и вернет ошибку (мы хотели посчитать количество уникальных типов устройств для каждого года):

```
SELECT
    DISTINCT type,
    c_year,
    COUNT(DISTINCT type) OVER (PARTITION BY c_year)
FROM devices;
```

РАНЖИРУЮЩИЕ ОКОННЫЕ ФУНКЦИИ

- В данной подглаве появятся пока неизвестные нам четыре функции - **ROW_NUMBER**, **RANK**, **DENSE_RANK**, **NTILE**. В принципе все что они делают - присваивают порядковый номер в контексте той группировки и сортировки, которая находится внутри конструкции **OVER**. Сортировку и группировку делать необязательно, но эти функции имеют смысл только вместе с ними, иначе это номер для всех и всегда будет равен единице. Несмотря на общую цель, каждая из перечисленных функций имеет свои особенности, которые мы сейчас рассмотрим.

- Приведем пример самой простой функции - **ROW_NUMBER**. Она делает точь в точь то, что написано в предыдущем пункте без всяких дополнений - присваивает номер согласно группе и сортировке. Например, мы применили группировку по типу (**PARTITION BY type**) - следовательно у нас образовалось три группы - с типом Computer, Tablet и Monitor. Здесь же мы применили сортировку по цене по возрастанию (**ORDER BY price ASC**). Значит в группе Computer устройство *Lenovo Legion T7* получит 1 номер (815 евро), *Dell Precision 3660* получит 2 номер (879 евро) и т.д. В свою очередь в группе Monitor устройство *Samsung Odyssey* получит 1 номер (227 евро), *LG UltraWide 49W* - 2 номер (289 евро) и т.д.

```
SELECT
    name,
    type,
    price,
    ROW_NUMBER() OVER(PARTITION BY type ORDER BY price ASC)
AS number
FROM devices;
```

name	type	price	number
Lenovo Legion T7	Computer	815.00	1
Dell Precision 3660	Computer	878.00	2
Acer Nitro 50	Computer	928.00	3
Apple Mac Mini M2	Computer	969.00	4
Samsung Odyssey	Monitor	227.00	1
LG UltraWide 49W	Monitor	289.00	2
Philips Envia 34M	Monitor	327.00	3
Nokia T20	Tablet	408.00	1
Microsoft Surface	Tablet	455.00	2
Huawei MatePad	Tablet	473.00	3

- Функция **RANK()** тоже присваивает номер каждой строке, но ведет себя немного необычно, если при сортировке встречается два одинаковых значения. В таком случае строкам присваивается одинаковый номер, а строке которой будет идти за ними - номер с пропуском следующего значения (т.е. первые две строки получат 1 номер, а третья - 3 номер). Чтобы увидеть эту особенность на примере, будем группировать устройства по их типу, а сортировать по году производства:

```
SELECT
    name,
    type,
    c_year,
    RANK() OVER(PARTITION BY type ORDER BY c_year ASC) AS
number
FROM devices;
```

name	type	year	number
Apple Mac Mini M2	Computer	2019	1
Lenovo Legion T7	Computer	2019	1
Acer Nitro 50	Computer	2021	3
Dell Precision 3660	Computer	2022	4
LG UltraWide 49W	Monitor	2019	1
Samsung Odyssey	Monitor	2022	2
Philips Envia 34M	Monitor	2022	2
Microsoft Surface	Tablet	2021	1
Huawei MatePad	Tablet	2021	1
Nokia T20	Tablet	2023	3

- Функция **DENSE_RANK()** имеет схожий с **RANK()** принцип присвоения номера строке, т.е. если при сортировке встречаются одинаковые значения, то им присваиваются одинаковые номера, но для следующей строки номерное значение не пропускается (т.е. первые две строки получат 1 номер, а третья - 2 номер). Повторим запрос из предыдущего пункта, но уже с **DENSE_RANK()**:

```
SELECT
    name,
    type,
    c_year,
    DENSE_RANK() OVER(PARTITION BY type ORDER BY c_year ASC)
AS number
FROM devices;
```

name	type	year	number
Apple Mac Mini M2	Computer	2019	1
Lenovo Legion T7	Computer	2019	1
Acer Nitro 50	Computer	2021	2
Dell Precision 3660	Computer	2022	3
LG UltraWide 49W	Monitor	2019	1

Samsung Odyssey	Monitor	2022	2
Philips Envia 34M	Monitor	2022	2
Microsoft Surface	Tablet	2021	1
Huawei MatePad	Tablet	2021	1
Nokia T20	Tablet	2023	2

- Последняя функция ранжирования – **NTILE** принимает единственный числовой параметр. С помощью этого параметра она условно делит сгруппированную группу (если группировка не указана, то всю таблицу) на соответственное количество частей и присваивает каждой строке номер той группы, в которую она попала (если предоставлена сортировка, то она учитывается при присвоении номера). Для разнообразия не будем группировать таблицу, но разделим нашу таблицу на три части, отсортированные по году создания:

```
SELECT
    name,
    c_year,
    NTILE(3) OVER(ORDER BY c_year) AS part
FROM devices;
```

name	c_year	part
Apple Mac Mini M2	2019	1
Lenovo Legion T7	2019	1
LG UltraWide 49W	2019	1
Microsoft Surface	2021	1
Huawei MatePad	2021	2
Acer Nitro 50	2021	2
Samsung Odyssey	2022	2
Philips Envia 34M	2022	3
Dell Precision 3660	2022	3
Nokia T20	2023	3

ОКОННЫЕ ФУНКЦИИ СМЕЩЕНИЯ

- Функции смещения позволяют заглянуть в следующие или предыдущие строки относительно текущей строки - в подавляющем большинстве случаев это необходимо только для сравнения.

- **FIRST_VALUE** и **LAST_VALUE** принимают в качестве параметра названия столбца и применяются для получения первого и последнего по величине значения (в контексте сортировки, если она применяется) **на данный момент** во всей таблице или в группе, если в конструкции **OVER** применяется группировка. Попробуем сгруппировать таблицу по типу устройства, а потом получить максимальную и минимальную цену в рамках конкретной группы. Заметим, что максимальное значение **cur_max_value** постепенно нарастает.

```
SELECT
    name,
    type,
    FIRST_VALUE(price) OVER(PARTITION BY type ORDER BY price)
AS cur_min_price,
    LAST_VALUE(price) OVER(PARTITION BY type ORDER BY price)
AS cur_max_price
FROM devices;
```

name	type	cur_min_value	cur_max_value
Lenovo Legion T7	Computer	815.00	815.00
Dell Precision 3660	Computer	815.00	878.00
Acer Nitro 50	Computer	815.00	928.00
Apple Mac Mini M2	Computer	815.00	969.00
Samsung Odyssey	Monitor	227.00	227.00
LG UltraWide 49W	Monitor	227.00	289.00
Philips Envia 34M	Monitor	227.00	327.00
Nokia T20	Tablet	408.00	408.00
Microsoft Surface	Tablet	408.00	455.00
Huawei MatePad	Tablet	408.00	473.00

- Функции **LAG** и **LEAD** позволяют получить значения предыдущей и последующей строки соответственно. Кроме названия столбца, значения которого мы хотим получить, можно также получить то количество строк, на которое мы хотим перескочить (по умолчанию - 1), а также то, значение, если значение строки невозможно получить. Повторим запрос из предыдущего пункта, но вместо **FIRST_VALUE**/**LAST_VALUE** применим **LAG**/**LEAD**:

```
SELECT
    name,
    type,
    LAG(price) OVER(PARTITION BY type ORDER BY price) AS
prev_price,
    LEAD(price) OVER(PARTITION BY type ORDER BY price) AS
next_price
FROM devices;
```

name	type	prev_price	next_price
Lenovo Legion T7	Computer	NULL	878.00
Dell Precision 3660	Computer	815.00	928.00
Acer Nitro 50	Computer	878.00	969.00
Apple Mac Mini M2	Computer	928.00	NULL
Samsung Odyssey	Monitor	NULL	289.00
LG UltraWide 49W	Monitor	289.00	327.00
Philips Envia 34M	Monitor	289.00	NULL
Nokia T20	Tablet	NULL	455.00
Microsoft Surface	Tablet	408.00	473.00
Huawei MatePad	Tablet	455.00	NULL