

## 4. Агрегатные функции и операторы

Агрегатные функции используются для объединения данных

Например, надо подсчитать кол-во актеров в фильме с идентификатором 384.

```
===== агрегатные функции =====

3. Подсчитайте количество актеров в фильме Grosse Wonderful (id - 384)
* Используйте таблицу film
* Соедините с film_actor
* Отфильтруйте, используя where и "title like" (для названия) или "film_id =" (для id)
* Для подсчета используйте функцию count, используйте actor_id в качестве выражения внутри функции
--Ф3, Аксиомы Армстронга

select count(fa.actor_id)
from film f
join film_actor fa on fa.film_id = f.film_id
where f.film_id = 384

=====

Results
+-----+
| count |
+-----+
| 4      |
+-----+
```

Теперь представим ситуацию, что помимо количества нам бы хотелось понять, что это за название фильма

Если просто добавим f.title - будет ошибка, т.к. столбец не инкапсулирован.

Инкапсуляция - т.е. что-то находится в чем-то, оно закрыто, оно спрятано.

Оператор `group by`

Когда вы работаете с агрегатными функциями, то все, что не инкапсулировано в агрегатные функции, должно быть перечислено в **group by**. Например `fa.actor_id` находится внутри агрегатной функции - все отлично, мы нигде дополнительно это указывать не должны. А вот `f.title` находится вне агрегатной функции, находится самостоятельно, сам по себе, мы должны его обязательно указать в группировке. И после этого наш запрос будет выполняться.

```
select f.title, count(fa.actor_id)
from film f
join film_actor fa on fa.film_id = f.film_id
where f.film_id = 384
group by f.title
```

FROM

	title	count
1	Grosse Wonderful	4

Более того, группировать лучше не по `f.title`, а по `f.film_id` (поскольку названия фильмов могут совпадать).

### Немного теории.

Когда вы работаете с группировкой, ваша первоочередная задача - стараться использовать функциональные зависимости.

Совет - почитайте аксиомы Армстронга ([статья в википедии на английском и ссылкой на книгу Уильяма В. Армстронга](#)). Они посвящены работе с функциональными зависимостями, избыточностями, транзакционными избыточностями и т.д. Это больше вопрос проектирования, архитектуры БД.

В соответствии с первой аксиомой и одним из первых законов реляционных баз данных - функциональная зависимость.

У вас есть `film_id`, он является первичным ключом (имеет ограничение первичного ключа). Согласно функциональной зависимости все значения кортежа (строки) зависят от первичного ключа. Другими словами для каждого первичного ключа существует свой строго уникальный набор значений в кортеже. Если мы добавим все столбцы из таблицы, то в группировке нам перечислять другие столбцы не нужно, достаточно `film_id`, т.к. все остальное функционально зависит от своего первичного ключа (от `film_id`).

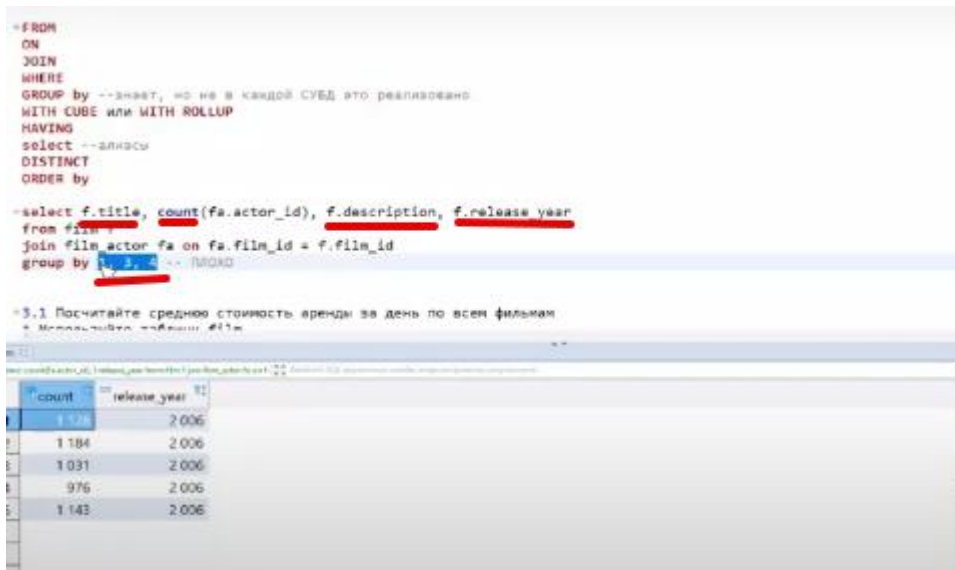
```
--select f.title, count(fa.actor_id), f.description, f.release_year
from film f
join film_actor fa on fa.film_id = f.film_id
group by f.film_id -- ХОРОШО

--select f.title, count(fa.actor_id), f.description, f.release_year
from film f
join film_actor fa on fa.film_id = f.film_id
group by f.title, f.description, f.release_year -- ПЛОХО
```

Но если стоит задача группировать по нескольким столбцам, естественно указываем в группировке оба столбца.

Если вам нужно получать какую-то сложную логику по группировке, то старайтесь эту логику группировать в запрос и делать подзапросом.

Интересный момент: PostgreSQL поддерживает нумерацию столбцов и их в группировке можно обозначить цифрами.



А вот **алиасы (псевдонимы)** в группировке использовать нежелательно, т.к. в PostgreSQL это сработает, в других базах - нет. Об этом надо знать и псевдонимы использовать при группировке нежелательно. Одновременно алиасы и нумерацию использовать можно.

Смысл агрегатных функций как раз в том, что неважно будет ли count, average, sum и пр., интерпретатор (сам SQL), имея 1000 строк, для каждой строки высчитывает это значение, получив его, суммирует и "схлопывает". Агрегатная функция объединяет данные, считается общий результат.

В целях создания меньшей нагрузки на сервер следует помнить о порядке выполнения: **where** - до группировки, **having** - после группировки.



Если вам нужно отфильтровать данные, надо это делать после группировки, т.к. это создает меньшую нагрузку на сервер.

Есть различные варианты группировки: не только по одному столбцу, а по несколько столбцов комплексно.

Берем идентификатор пользователя, идентификатор сотрудника и сумму, которую заплатил пользователь. В группировке указываем id пользователя и id сотрудника. Условие - id пользователя < 5. Можем ли мы в этом запросе получить информацию, сколько каждый пользователь заплатил одному и второму сотруднику.

```

select c.customer_id, p.staff_id, sum(amount)
from payment p
join customer c on p.customer_id = c.customer_id
where c.customer_id < 5
group by grouping sets (c.customer_id, p.staff_id)

```

===== подзапросы =====

5. Выведите количество фильмов, со стоимостью аренды за ден

customer(\*)

```

select c.customer_id, p.staff_id, sum(amount) from payment p join customer c

```

	customer_id	staff_id	sum
1	1	2	53,85
2	1	1	60,85
3	2	2	67,88
4	2	1	55,86
5	3	2	70,88
6	3	1	59,88
7	4	1	49,88
8	4	2	31,9

```

select c.customer_id, p.staff_id, sum(amount)
from payment p
join customer c on p.customer_id = c.customer_id
where c.customer_id < 5
group by grouping sets (c.customer_id, p.staff_id)

```

customer(\*)

```

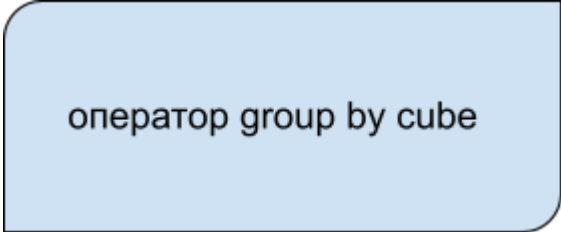
select c.customer_id, p.staff_id, sum(amount) from payment p join customer c

```

	customer_id	staff_id	sum
1	1	[NULL]	114,7
2	2	[NULL]	123,74
3	3	[NULL]	130,76
4	4	[NULL]	81,78
5	[NULL]	1	226,47
6	[NULL]	2	224,51

оператор grouping sets

**Grouping sets** - по каждому переданному параметру будет производиться своя собственная группировка и выводиться в результат. Сперва получаем результат, группируя по пользователю, потом вывели информацию по суммам платежей, которые принял каждый сотрудник. Представьте, что вы - федеральная сеть, есть множество данных, одним таким запросом вы можете получить подробную статистику.



оператор group by cube

Если хотим чего-то большего и более сложного, используем **group by cube** - мы из наших данных формируем кубик и для каждой грани получаем свой результат.

Можем увидеть, сколько:

- первый пользователь заплатил первому сотруднику;
- первый пользователь заплатил второму сотруднику;
- первый пользователь заплатил всего;
- второй пользователь заплатил первому сотруднику;
- второй пользователь заплатил второму сотруднику;
- второй пользователь заплатил всего;
- каждый сотрудник получил индивидуально (сумму);
- какова была вообще общая сумма.

Вот такие своеобразные подитоги.

```

select c.customer_id, p.staff_id, sum(amount)
from payment p
join customer c on p.customer_id = c.customer_id
where c.customer_id < 5
group by cube (c.customer_id, p.staff_id)
order by 2

```

	customer_id	staff_id	sum
1	1	1	60,85
2	1	2	53,85
3	1	[NULL]	114,7
4	2	1	55,86
5	2	2	67,88
6	2	[NULL]	123,74
7	3	1	59,88
8	3	2	70,88
9	3	[NULL]	130,76
10	4	1	49,88
11	4	2	31,9
12	4	[NULL]	81,78
13	[NULL]	1	226,47
14	[NULL]	2	224,51
15	[NULL]	[NULL]	450,98

Оператор всегда выведет общий результат. Группировка формирует полноценный куб. Этот промежуточный результат, с которым вы будете работать дальше, оборачиваете в подзапрос. Не нужно делать отдельные группировки по сотрудникам и пр. Для вывода информации обращаетесь к этому подзапросу. Вам не нужно делать 10 запросов - делаете 1 запрос, получаете данные, а потом отсюда получаете все, что вам необходимо. Это очень удобно для отчетов.

оператор rollup

Есть третий вариант группировки, он используется крайне редко. Это **rollup**. Сгруппируем по месяцу.

**Rollup** будет уменьшать количество действий.

Формирует пирамиду (в отличие от куба при группировке cube). На каждой итерации для каждого последующего действия он уменьшает количество действий. Если используем **cube** - получаем 59 записей, если используем **rollup** - 36 записей: все

комбинации для пользователя, все комбинации минус один для каждого сотрудника и соответственно на каждый идентификатор сотрудника и пользователя будет минус одна комбинация по месяцу. Получается, что формируется не кубик, а скошенная пирамида - на каждый последующий аргумент убирается одно действие.

Совет: почитайте статьи на хабре, чтобы углубиться в понимание **rollup** ([например](#)).