Типичные ошибки в запросах:

1. Неправильный порядок или ошибки в ключевых словах.
2. Неправильно названы используемые в запросе функции и операторы (например, DATEPART, а не DATE_PART).
3. Неправильно указаны имена столбцов.
4. Неправильно выполнена сортировка записей.
5. Неправильно проведены расчёты.
6. Пропущена запятая при перечислении столбцов в SELECT
7. Лишняя запятая после имени последнего столбца в SELECT
8. Не закрыты скобки (проверьте, что количество открывающих скобок равно количеству закрывающих).
9. Допущена ошибка в подзапросе (перед выполнением всего запроса проверьте, что работают отдельные подзапросы).
10. Запущены сразу несколько запросов, не разделённые точкой с запятой.
11. Кавычки должны быть одинарные ‘ ‘ а не двойные “ “
12. Разделить в числах точка, а не запятая (1.5)
13. При **делении** всегда одно из значений должно приводиться к :: DECIMAL

Redash:

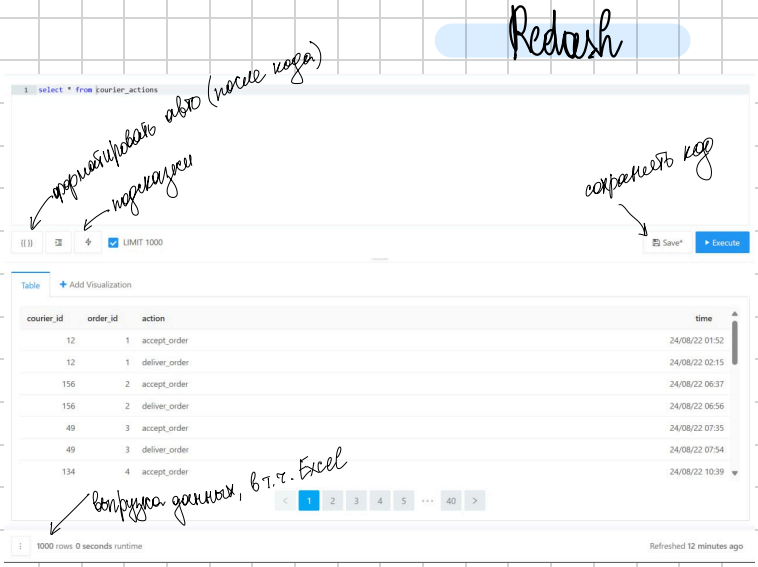
• по умолчанию **дата** отображается как дд/мм/гг час/мин (год 2 последн цифры, секунд нет) и не важно как дата на самом деле записана в табл (если секунд не видно, не значит что их нет). Если запись с секундами, а в запрос мы пишем без –> выдаст ошибку

• чтобы см **как данные выглядят исходно**, нужно перевести в текст: SELECT time : : TEXT FROM orders

в табл, а не так, как видел Redash

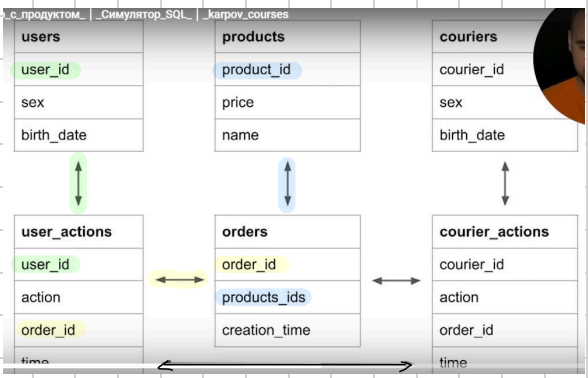
• тоже самое **знаки после запятой** – redash сокращает до 2ух, но не факт, что их 2. Округление нужно делать прям в коде (ROUND) или см в экселе, не просто на то, что выдает redash

• **формат даты**, который нужно использовать в функциях: ‘YYYY–MM–DD HH:MM:SS’
*берем в кавычки (напр. при исп. BETWEEN)



продуктом | _Симулятор_SQL_ | _karpov_courses	A	B	C
1	**кто**	**что сделал**	**когда**
2	пользователь №1	создал заказ №1	13 августа 15-45
3	курьер №1	принял заказ №1	13 августа 15-55
4	курьер №1	доставил заказ №1	13 августа 16-25
5			
6	**заказ**	**товары**	
7	№1	хлеб, чай, сок	
8			
9	**пользователь**	**возраст**	**пол**
10	№1	31	M
11			
12	**курьер**	**возраст**	**пол**
13	№1	25	M

– данные хранятся не в такой табл, а в **БД** (для удобства, в т. ч. из–за объема)



структура/схема БД (в виде табл.)
стрелки означ. связь между табл., что можно «прыгнуть» из одной в др (не всегда за 1 шаг)

Типы данных

Типы данных (в табл., для PostgreSQL)

Тип данных	Описание	Пример
INT	Целое число	id пользователя: 132
NUMERIC / DECIMAL	Вещественное число	Стоимость товара: 120.55
VARCHAR	Текст	Действие с заказом: «create_order»
DATE	Дата с точностью до дня	Дата рождения пользователя: 25/03/91
TIMESTAMP	Дата с точностью до секунды	Время регистрации в приложении: 24/08/22 01:52:24
[]	Массив	Список id товаров в заказе: [1, 13, 22]

Boolean: true/false/null
boolean/bool to declare a column
- inserting data into boolean column:
  1/yes/y/t/true –> true
  0/no/false/f –> false
- selecting data from a boolean column:
  t –> true
  f –> false
  space –> null

Numeric, 2 types:
1. Integers
   1.1 SMALLINT – 2-byte, signed integer (-32,768; 32,768)
   1.2 INT – 4-byte (-2,147,483,648; 2,147,483,648)
   1.3 SERIAL
2. Floating-point number
   2.1 float(n) – precision: (n; 8-byte)
   2.2 real/float8 – 4-byte
   2.3 numeric/numeric(p,s) – real number with p digits and s number after the decimal point; is the exact number

Temporal – date/time data
1. DATE
2. TIME
3. TIMESTAMP – both date and time
4. TIMESTAMPTZ – timezone-aware timestamp
5. INTERVAL – periods

Character, 3 types:
1. CHAR (n) – fixed length character with spaces inserted
   - if I insert string which is shorter –> will add spaces
   - string is longer –> error
2. VARCHAR (n) – variable-length
   - stores up to n characters
   - string is shorter –> no spaces inserted
3. TEXT – variable-length; unlimited length

Array
- array of strings/integers..in array columns
- e. g. storing days of the week/month of the year

JSON – requires reparsing for each processing ?
JSONB – json data in binary format

UUID – stores Universal Unique Identifiers defined by RFC 4122
- used to store (hide) sensitive data

*EPOCH – value from a daytime value, i. e. number of seconds since 1970-01-01 00:00:00 UTC

Урок 3. Карпов

Операторы Sql

Postgre SQL – хранилище данных
SQL – язык запросов, с помощью кот. мы обращаемся к данным

Порядок написания операторов в запросе:

Sir Filip Will Go Home Ordinarily
SELECT –– перечисление полей результирующей таблицы
FROM –– указание источника данных (БД)
JOIN –– соединяет таблицы, можно обращаться к обеим
WHERE –– фильтрация данных
GROUP BY –– группировка данных
HAVING –– фильтрация данных после группировки
ORDER BY –– сортировка результирующей таблицы
LIMIT –– ограничение количества выводимых записей

Порядок выполнения запросов (реальный):

Funny Jokes Will Get Harry Smiling Out Loud
FROM – выбирается нужная табл.
JOIN
WHERE – отбираются строки, соотв. условию
GROUP BY – строки объед. в группы, происх. агрегация
HAVING – фильтр как WHERE но идет после GROUP BY
SELECT – отбираются нужные столбцы, применяются ф–ии
ORDER BY – сортировка полученной табл.
LIMIT – ограничивается кол–во выводимых записей

SELECT column – перечисляем что мы «вытаскиваем» из табл. Просто показывает что попадет в конечную таблицу
SELECT* – достать сразу все столбцы
• лучше (этичнее) писать не *, а перечислять все колонки. Запросы должны быть **явные**, команды капсом

WHERE – условие отбора, выведет только те строки, которые соответствуют условию

```
SELECT action, courier_id, order_id, time
FROM table
WHERE action = 'deliver order' AND (courier_id = 12 OR courier_id = 100)      или
WHERE action = 'deliver order' AND courier_id IN (12, 100, 450)
```

• сначала происходит отбор строк по WHERE, только потом действия, указанные в SELECT
• можно сравнивать с временем и датой. Важно что б в табл. были даты, а не текст в виде даты. time > '06–09–22 00:00:00' (9 июня)
• в WHERE нельзя писать алиас (AS), нужно в SELECT (учит. последовательность выполнения запроса)
• если в WHERE исп. AND, то sql не совмещает 2 условия, а выполн. их по очереди
• если нет скобок (приоритета), то сначала выполняется AND, затем OR

```
WHERE
order_id IN (SELECT order_id FROM table WHERE action = 'deliver') AND
            (SELECT order_id FROM table ORDER BY time LIMIT 100)
```

– выдаст все последние 100 записей, а не только те, что 'deliver'

—

```
WHERE
order_id IN (SELECT order_id FROM table WHERE action = 'deliver') AND
order_id IN(SELECT order_id FROM table ORDER BY time LIMIT 100)
```

– выдаст 100 последн., кот. deliver

• WHERE action <> 'cancel_order' – в рез–тат могут попасть отмененные заказы, поскольку такой заказ исключит только строки с 'cancel_order' а другие действия по заказу оставит (исключит только строки 4 и 6)

```
WHERE order_id NOT IN (SELECT order_id FROM table WHERE action = 'cancel_order')
```

– запрос удалит все что связано с отмен. заказами (исключит строки 3, 4, 5, 6)

• в WHERE нельзя просто сравнивать 2 подзапроса, тк выдаст ошибку 'subquery returns more than 1 row', нужно заменить на:
Сравни номера заказов в courier_actions и orders: courier_actions.order_id = orders.order_id
общий вид записи **таблица.столбец**

• нельзя исп. агрег. функции в WHERE, тк sql работает в др. порядке.
нет: WHERE price > AVG(price)
да: WHERE price > (SELECT AVG(price) FROM table)

row_numb	order_id	action
1	1	create
2	1	deliver
3	2	create
4	2	cancel
5	3	create
6	3	cancel

WHERE _ IN _ – входит ли значение в список
WHERE column_1 IN (‘product_1’, ‘product_2’, ‘product_3)
WHERE _ NOT IN _ – противоположно

WHERE _ BETWEEN _ AND _ – входит ли значение в интервал (границы интервала включаются)
WHERE column_2 BETWEEN 5 AND 10
• работает с временем/датой:
WHERE column_3 BETWEEN ‘2022-11-20’ AND ‘2022-12-31’ (2022.12.31 00:00:00, т е не попадут записи после полуночи)
или WHERE column_2 >= 5 AND column_2 <= 10
WHERE _ NOT BETWEEN _ – противоположно

дата меньше
дата больше

WHERE column **IS NULL** – проверка на нулевые (пустые значения); вернет все пустые
WHERE column **IS NOT NULL** – вернет непустые
• любое сравнение с NULL выдаст NULL

GROUP BY – группировка

1. группировка может проводиться по нескольким критериям сразу

```
SELECT column_1, column_2, SUM(column_3)
FROM table
GROUP BY column_1, column_2
```

2. к группам, кот. образовались с помощью GROUP BY можно применять сразу неск. агр. ф-ий (в т. ч. к разным колонкам)

```
SELECT column_1, SUM(column_2), AVG(column_3)
FROM table
GROUP BY column_1
```

3. после группировки к рез-там агр. ф-ий можно применять другие функции

```
SELECT column_1, SUM(column_2) :: DECIMAL / SUM(column_3) * 100
FROM table
GROUP BY column_1
```

4. агрегацию не обязательно проводить по уже имеющимся колонкам

```
SELECT column_1, SUM(some_func(column_2))
FROM table
GROUP BY column_1
```

5. группировку можно делать по новым полям, только что подсчитанным в SELECT.
В GROUP BY можно использовать алиас из SELECT:

```
SELECT DATE(column_1) AS date, SUM(column_2) AS sum
FROM table
GROUP BY date
```

*так исп. алиасы можно только в PostgreSQL (в обход порядка выполнения ф-ий). в др. нельзя, поэтому лучше в GROUP BY исп. не алиас, а дублировать инф-ю из SELECT

6. если не указывать агрег. фу-ию, а только GROUP BY, то рез-тат будет такой же как SELECT DISTINCT (выдаст уникальные значения)

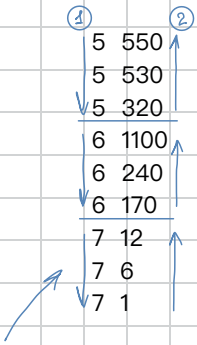
```
SELECT user_id
FROM user_actions
GROUP BY user_id
=
SELECT DISTINCT user_id
FROM user_actions
```

7. Все неагрегированные колонки (те, над кот. не исп. агрегирующая ф-ия), кот. есть в SELECT должны быть и в GROUP BY
*но если колонка есть в GROUP BY, ее не обязательно указывать в SELECT (тк GROUP BY выполняется первее)
*те колонки над которыми проводится агрегация, наоборот, не могут быть указаны в GROUP BY
8. в GROUP BY и ORDER BY можно писать не названия, а номера колонок из SELECT

```
SELECT column_1, column_2, SUM(column_3)
FROM table
GROUP BY 1,2
ORDER BY 3
```

9. ЕСЛИ с GROUP BY использовать CASE, то CASE выполняется первее (вопреки порядку)
10. можно фильтровать с помощью CASE, WHERE, HAVING, FILTER (выполняются по очереди соответственно)
11. число строк в результирующей табл. = кол-во групп (по кот. группировалось) в исходной табл.

HAVING – такой же принцип как WHERE, но выполн. после группировки
- фильтрует рез–таты, кот. уже сгруппированы
- нельзя исп. алиасы из SELECT
- необязательно указывать колонки из SELECT (в отлич. от GROUP BY)
- не работает без GROUP BY



ORDER BY column – сортировка
ORDER BY user_id ASC – сортировка по возрастанию и алф. порядок
ORDER BY user_id – то же самое
ORDER BY user_id DESC – по убыванию
SELECT action, courier_id, order_id, time FROM table ORDER BY user_id LIMIT 10
- можно указывать неск. столбцов для сортировки (отсортирует первую колонку, затем внутри ее категорий отсортирует знач. 2ой)
- извлечь все столбцы из табл., сначала отсортировать по 1ому столбцу по убыванию, затем по 2му по возрастанию

SELECT * FROM table ORDER BY column_1 DESC, column_2
SELECT * FROM table ORDER BY name, action, time DESC – сортировка сначала name, action по возр., затем time по убыв.
- внутри ORDER BY можно проводить арифметические расчеты (?)

ORDER BY delivery_time – creation_time DESC

LIMIT 10 – вывести только первые 10 строк
- всегда ставить лимит, ибо всю таблицу вывести почти невозможно

SELECT action, courier_id, order_id, time FROM table LIMIT 10

AS – алиас, колонки можно переназвать, чтобы они выводились под другим названием
SELECT name AS product_name, price AS product_price – в оригинале были name и price, выведет product_name и product_price
- можно AS вообще не писать: SELECT column product_name (в рез–те переименует в product_name)

Фильтрация. Что выбрать?
- WHERE – фильтрация строк; исп. если фильтрация нужна до группировки (GROUP BY)
  - ✘ не работает. с агрег., группировк., и алиас
- HAVING – фильтрация агрегатов; исп. если фильтрация нужна после группировки (поверх сгруппированных данных)
  - ✘ нет смысла без GROUP BY
- в SELECT нужно указывать только то что в GROUP BY (не важно что в HAVING)
- FILTER (WHERE_) происходит в последнюю очередь (?)
- агрегация, фильтрация и группировка происходят до SELECT
- проверка наличия в др. табл.:
  - EXISTS – если много данных, останавливается при 1ом совпадении
  - IN – если мало данных, (может тупить если много) проверяет входит ли в список

Что работает со всей колонкой, а что только с 1им значением:
- если в коде исп. …, то работа над 1им значением
  - GROUP BY
  - HAVING
  - FILTER (WHERE..)
  - агрегирующие ф–ии

*SQL не знает как агрегировать данные (avg) без группировки
- WHERE и обычные функции (не sum/avg..) работают над всей колонкой, а не над 1им знач.

Агрегация данных

Агрегирующие функции – те, которые обрабатывают несколько строк и выдают одно обобщенное значение.

- SQL не знает как агрегировать данные (avg) без группировки GROUP BY
- Пишутся в SELECT:

```
SELECT COUNT (column) AS hello
FROM table
```

COUNT (column) – считает кол–во всех не NULL значений (не уникальных)
COUNT (*) – кол–во всех значений в таблице (в т. ч. NULL)
COUNT (DISTINCT column) – кол–во уникальных значений

SELECT DISTINCT column – выдаст только уникальные значения (без повторяющихся)
SELECT DISTINCT column, column_2, column_3.. – выдаст уникальные комбинации из всех перечисленных колонок

SUM (DISTINCT column) – расчет будет производиться только по уникальным значениям
SUM (column) – сумма всех значений

MAX (column) – максимальное значение

MIN (column) – минимальное

AVG (column) – среднее значение

- не все работают с текстом/датой/временем – тк. что такое сумма текста?

Агрегатные выражения с фильтрацией

FILTER (WHERE_)

- как WHERE, просто пишется в SELECT
- исп. только вместе с агрегатной функцией (если нужно не 1 знач. (агр), а столбец значений, то исп. CASE)
- нельзя: SELECT order_id FILTER (WHERE..)
  можно: SELECT MAX(order_id) FILTER (WHERE..)
- на вход агрег. ф–ии будут поданы только те знач., кот. истина

```
SELECT agg_function(column) FILTER (WHERE condition)
FROM table
```

```
SELECT AVG(price) FILTER (WHERE category = 'fish') AS цена
FROM table
```

- агрег. ф–ии можно совмещать с обычными

```
SELECT column_1, MIN(DATE_TRUNC('month', column_2))
FROM table
GROUP BY column_1
```

- важен порядок применения ф–ий. Разный порядок = разный рез–тат (но не всегда)

```
SELECT column_1, MIN(DATE_PART('month', column_2))
FROM table
GROUP BY column_1
```

не равно

```
SELECT column_1, DATE_PART('month', MIN(column_2))
FROM table
GROUP BY column_1
```

WHERE если нужно сложить/мин/посчитать.. не все значения, то можно исп. WHERE. порядок ф–ий: FROM –> WHERE –> SELECT

```
SELECT COUNT(column) AS count
FROM table
WHERE column_2 > 100
```

- внутри WHERE нельзя писать агрегирующую функцию (MIN/MAX..) –> исп. подзапрос

```
WHERE column = (SELECT MAX(column) FROM table)
```

- можно исп. арифметику: WHERE column = (SELECT MAX(column) FROM table) – 100

Функции

- кавычки везде одинарные ‘ ‘ а не двойные “ “
- слова без кавычек – это название колонки; слова в кавычках – это текст

Арифметические ф–ии: + – * / % ^

- нельзя написать price/120%, нужно price/1.2

SELECT 7500/75 –> 100
SELECT column – 100 –> вычтет 100 из каждого значения в колонке
SELECT (column_1 + column_2) / 2 –> среднее арифметическое из 2ух колонок

- при делении хотя бы 1 из значений нужно привести к ::DECIMAL

Функции сравнения:

=		– равно
<> или !=		– не равно
<	>	– меньше, больше
<=	>=	– меньше или равно, больше или равно

- при сравнениях можно использовать:

AND
OR
NOT

- Ответы:

TRUE
FALSE
NULL – когда одно из сравниваемых NULL

Приоритет выполнения операций:

1) то, что в скобках
2) умножение и деление (* и /)
3) сложение и вычитание (+ и –)
4) операторы сравнения (=, !=, >, <, >=, <=)
5) NOT
6) AND
7) OR

Изменить тип данных, 2 способа:

1) **CAST** (column/“text” AS data type):
SELECT CAST (column_1 AS VARCHAR)

2) SELECT column : : **VARCHAR**

Превратить время+дата в дату, 3 способа:

*time – назв. колонки
1) SELECT DATE(time)

2) SELECT CAST (time AS DATE)

3) SELECT time: :DATE

LENGTH (column/“text”) – выводит кол–во знаков

определите товар с самым длинным названием в табл. products. Выведите его наименование, длину, наименование в символах, цену товара. Колонку с длиной наименования назовите name_length.

```
SELECT name, price, LENGTH(name) AS name_length
FROM products
ORDER BY name_length desc
LIMIT 1
```

UPPER (column/“text”) – выводит значение капсом

LEFT (column/“text”, n) – выводит первые n символов в строке

SELECT UPPER(LEFT(“text,3)) AS new_name => TEX

SPLIT_PART (“text”, “разделитель”, n) – разделяет текст по разделителям и выводит n часть:

SPLIT_PART (“text.new.year”,”.”,2) => new

ROUND (number, число знаков после запятой) – если не указать число знаков, по умолчанию округлит до целого числа.

- Redash выдаст 1.5 как 1.50. Изменить можно в edit visualisation

CONCAT (‘smth’, ‘smth’, ‘smth’..) – «склеивает» куски и превращает их в текст. Работает если «куски» можно конверт. в текст

если нужно вывести текст в формате:

```
SELECT CONCAT(‘Кампания №’, ads_campaign) AS ads_campaign
```

*ads_campaign описывается ниже; № кампании присваивается через CASE (см задачник 2.2.2.1)

ads_campaign	cac
Кампания №1	55
Кампания №2	67

DATE_PART (part, column/конкретная дата или время) – извлекает часть из даты

- parts; ‘year’/‘month’/‘day’/‘dow’/‘isodow’/‘hour’
- part ‘dow’ – день недели (0–вскр, 6–суббота)
- part ‘isodow’ – наименование дня недели

DATE_PART (‘year’, date_birth)

DATE_TRUNC (‘part’, column) – округление времени/даты

- работает с данными типа TIMESTAMP или INTERVAL, ответ в том же формате
- вместо part указываем year/month/day/hour
- в ответе «менее значимые» части даты приравниваются к 0 (или 1 если это номер дня или месяца)

TO_CHAR (expression, format) – преобразует числовые / временные / датированные знач. в заданный формат (по шаблону):

TO_CHAR (current_date, ‘DD–MM–YYYY’) –> 11–02–2025

TO_CHAR (12345.678, ‘99999.99’) –> 12345.68

TO_CHAR (TIMESTAMP ‘2022–08–29’, ‘Dy’) –> выдаст день недели ‘Mon’

*если затем сортировать по получ. значению, то сортировка будет неверной –> будет сортировать лексически. Чтобы исправить:

TO_DATE((TO_CHAR(column1, ‘MM/YY’)), ‘MM/YY’) или DATE(column)

EXTRACT (part FROM date) – достает часть даты из даты

EXTRACT (month FROM ‘2017–06–15’)

- part может быть чем угодно: microsecond / second / minute / hour / day / week / month / quarter / year..
- вместо date можно

INTERVAL ‘date’ – для работы со временем, напр., вычесть неделю из тек. даты
SELECT NOW() – INTERVAL ‘1 year 2 month 1 week’

CASE – аналог IF/IFS в экселе, не просто отбирает значения (как WHERE), а что-то с ними делает (то что указано после THEN)
• пишется в SELECT (как одна из переменных, те нужны запятые)
• если logical_expression true –> выведет expression, в противном случае выведет expression_else
• если не указать else –> выведет null

```
CASE
WHEN logical_expression_1 OR logical_expression_1.1 OR .. THEN expression_1
WHEN logical_expression_2 THEN expression_2
...
ELSE expression_else
END AS case_example – AS добавлять не обязательно
```

• может помещаться внутрь агрегирующей ф-ии:

```
AVG (
    CASE
    WHEN .. THEN..
    ELSE
    END)
```

• в ELSE нельзя написать price = 0, нужно просто 0, тк первое подразумевает присвоение нуля
• нельзя присваивать алиасы
WHEN logical_expression THEN expression_1 AS new_expression –> не сработает

больш. меньш.
AGE (‘1st date’, ‘2nd date’) – показывает возраст в количестве дней (3678)
• 1st и 2nd date должны быть в формате TIMESTAMP
• ответ (разница) получается в формате INTERVAL
SELECT AGE (‘2022-12-12’, ‘2021-11-10’)

• если в 1st date ничего не указать, то подставится текущая дата
SELECT AGE (‘2021-11-10’)
или
(?) SELECT AGE(TIMESTAMP ‘2021-11-10’)
или
SELECT AGE(current_date, ‘2021-11-10’)

• для отображения в годах + мес + днях
SELECT AGE(current_date, ‘2021-11-10’) :: VARCHAR

• для отображения в годах:
DATE_PART(‘year’, AGE(birth_date)) : : INTEGER

LIKE – возвращает то, что содержит условие (напр., все, что содержит «чай», не важно, в середине или нет)
• возвращает TRUE/FALSE в зависимости от соответствия шаблону
Шаблон:
• _ – один символ
• % – любая последовательность символов (в т. ч. пустая)
SELECT ‘text.privet’ LIKE ‘text_’ –> false
SELECT ‘text.privet’ LIKE ‘text%’ –> true
SELECT ‘text.privet’ LIKE ‘%text%’ –> true
SELECT ‘text.privet’ LIKE ‘_._’ –> false
SELECT ‘text.privet’ LIKE ‘%.%’ –> true
• чувствителен к регистру: SELECT ‘text.privet’ LIKE ‘Text%’ –> false

NOT LIKE – противоположно

IN – показывает, входит ли значение в промежуток

- правильный синтаксис: WHERE column IN (value1, value2, value3)
- нельзя написать WHERE action IN action = ‘deliver_order’, тк это не промежуток

NOT IN – противоположно

COALESCE – выдает первое не null значение, может заменить его. Все части должны быть одного формата(?)

COALESCE (null, 1, ‘text’) –> 1

COALESCE (column, ‘text’) –> заменит null значения в колонке на ‘text’

Функции массивов:

ARRAY_LENGTH (array_column, 1) – показывает кол–во элементов (длину) в массиве

1 – размерность массива, по кот. считается его длина

SELECT ARRAY_LENGTH (cities, 1)

массив – несколько склеенных столбцов?

country	cities	array_length
UK	London, Stretford	2
SG	Singapore	1
US	Southlake, San Francisco, Brunswick, Seattle	4
CN	Beijing	1

UNNEST (array) – разворачивает массив и превращает его в набор строк (каждой строчке соотв. значение, входящее в массив).

SELECT UNNEST (array [‘one’, ‘two’, ‘three’] –>

one
two
three

ARRAY_AGG (array) – сворачивает значения в массив. Обратное от unnest

- “схлопывает” данные в том порядке, в кот. они поступают
- можно применять сортировку прям внутри функции: ARRAY_AGG (column ORDER BY column2 DSC)

ARRAY [column1, column2..] – превращает в массив

Подзапросы

Подзапросы – временная таблица, которая формируется для дальнейших расчетов и нигде не сохраняется.

- выдает только 1 значение. Если результат – целая колонка, то выдаст ошибку
- оформляется в скобки ( )
- может быть несколько подзапросов в одном запросе
- может быть подзапрос внутри подзапроса, матршека (тогда сначала выполн. самый внутренний, затем выше..)
- выполняется первее, чем основной запрос

- чаще в секции фильтрации (любой), но **может быть написан в**:

1. **FROM** (вместо указания таблицы):

```
SELECT column_1
FROM (
    SELECT column_1, column_2
    FROM table
) AS subquery_1;
```

– при таком синтаксисе (где в FROM есть только подзапрос) в основном запросе в SELECT можно достать только то, что является результатом подзапроса, а не любую колонку из таблицы, к кот. обращается подзапрос

– обязательно присваивать имя, иначе не сработает

– в конце ; иначе не сработает

2. **SELECT** (если запрос возвращает только 1 знач., напрм. рез–тат агрегир. ф–ии)

```
SELECT column_1, (SELECT MAX(column_1) FROM table) AS max_column
FROM table
```

при таком подзапросе будет выведена колонка column_1 и напротив каждого ее значения будет выведен рез–тат MAX (те сформируется 2ой столбец max_column

3. **WHERE** и **HAVING** (если запрос возвращает 1 столбец с 1 или неск. знач.)

4. **CASE** (продвинутые конструкции)

Почему удобно:

- может обращаться к др табл. (не к той, к кот. обращается основной запрос)
- написала огромный код, взяла его в скобки (сделала подзапросом), перед скобками пишу SELECT MAX
- нельзя сразу же обращаться к только что созданной колонке (алиас), выдаст ошибку. Нужно исп. подзапрос

CTE

WITH

• создает табличные выражения CTE (common table expressions) – временные табл., существующие только для 1го запроса.
• исп. для того, чтобы разбить сложный запрос на неск. частей
• всегда вызывается через подзапрос, а не просто алиасом. алиас CTE – имя подзапроса, а не алиас таблицы
• похоже на присваивание переменных
• если к подзапросу приходится обращаться неск. раз (дублировать его), то лучше вынести его в начало кода в WITH и работать как с переменной

```
WITH
subquery_1 AS (
    SELECT column_1, column_2, column_3
    FROM table
),
subquery_2 AS (
    SELECT column_1, column_2
    FROM subquery_1
)

SELECT column_1
FROM subquery_1
```

```
WITH
subquery_1 AS (
    SELECT column_1, column_2
    FROM table
)

SELECT column_1
FROM subquery_1
```

• если мы обращаемся к WITH в WHERE, то нельзя просто обратиться к алиасу в WITH (subquery_1), нужно вызвать всю табл. через подзапрос:

```
WITH
subquery AS (
    SELECT column_2
    FROM table_2
)

SELECT column_1
FROM table_1
WHERE column_1 IN (SELECT* FROM subquery)
```

Join

*по дефолту (если не указать какой), применится INNER JOIN

• **как работает:** join делает декартово (прямое) произведение 2ух множеств – всевозможные пары исходных строк. Затем оставляет только те, что соотв. условию в ON, затем добавляет те, строки, кот. должны остаться по гарантии

• 2 таблицы можно связать, если у них есть «связующее звено» key (колонка), которое нужно указать ON table1.key = table2.key

• в качестве key лучше брать столбцы, по которым можно однозначно идентифицировать строку/сущность (напр., id товара)

• джоины отличаются «гарантиями» – какие данные мы гарантировано сохраним

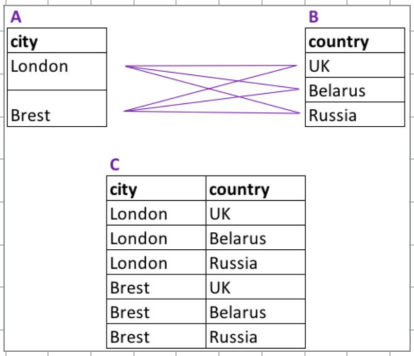
• если order_id совпадают (не факт что уникальны), то LEFT и RIGHT JOIN дадут одинаковый результат

• если order_id уникальны (не повт. в неск. строк) и совпадают в обеих табл., то все 4 JOIN дадут одинаковый результат

• джоинить массивы лучше после их раскрытия (ф–ия UNNEST)

• фильтровать лучше до объединения таблиц, тк это делает запрос быстрее

• лучше всегда и везде уточнять откуда берется колонка (таблица.колонка)



• **общая форма:**

```
SELECT column, column, column
FROM table
    ___ JOIN table
    ON table.column = table.column
    ___ JOIN table
    ON table.column = table.column ...
```

• джоинить можно не только табл. + табл., а табл. + подзапрос:

```
SELECT*
FROM table
    LEFT JOIN (SELECT order_id, UNNEST (product_ids) AS product_id FROM orders) t1
    ON table.order_id = t1.order_id
```

• в джоинах можно исп. алиасы если назв. табл. слишком большие:

```
SELECT a.column1, b.column2
FROM table1 AS a
    JOIN table2 AS b
    ON a.id = b.id
```

• если имя key совпадает в обеих табл., то можно исп. сокращенную запись через USING:

```
SELECT a.column1, b.column2
FROM table1 AS a
    JOIN table2 AS b
    USING (id)
```

LEFT (OUTER) JOIN – гарантирует, что данные из левой табл. точно попадут в финальную (а также, по возможности, добавится вся подходящая инф–я)

• алгоритм: 1) выполн. INNER JOIN; 2) добавл. строки, кот. не соотв. усл. ON, но есть в левой табл.

• исп. чаще всего

• не гарантирует, что левая таблица будет выглядеть без изменений, а гарантирует, что она вся попадет в финальную

• как работает: берет 1ый order_id из левой табл., идет в правую табл. и для каждой строки проверяет условие равенства. Если условие равенства выполняется (может не 1 раз), то записывает эту строку в финальную

```
SELECT*
FROM t1 LEFT JOIN t2 ON t1.order_id = t2.order_id
```

user_actions as t1

order_id	user_id
592	500
593	501

courier_actions as t2

order_id	courier_id
592	148
594	57

→

order_id	user_id	order_id1	courier_id
592	500	592	148
593	501		

=> если в левой табл. было 100 записей, это не значит, что в финальной тоже получится 100:

order_id	user_id
592	500

order_id	courier_id	action
592	148	accept_order
592	148	deliver_order

→

order_id	user_id	order_id1	courier_id	action
592	500	592	148	accept_order
592	500	592	148	deliver_order

RIGHT JOIN – гарантирует, что все из правой таблицы точно попадет в финальную (а также добавится вся подходящая инф-я)

- алгоритм: 1) выполн. INNER JOIN; 2) добавл. строки, кот. не соотв. усл. ON, но есть в правой табл.
- как LEFT JOIN, но наоборот

```
SELECT*
FROM t1 RIGHT JOIN t2 ON t1.order_id = t2.order_id
```

user_actions as t1

order_id	user_id
592	500
593	501

courier_actions as t2

order_id	courier_id
592	148
594	57

→

order_id	user_id	order_id1	courier_id
592	500	592	148
		594	57

(INNER) JOIN – выведет только те данные, которые есть в обеих таблицах (только пересечение)

- симметричен – не важно, в каком порядке записаны табл. (какая слева, а какая справа)

```
SELECT*
FROM t1 INNER JOIN t2 ON t1.order_id = t2.order_id
```

user_actions as t1

order_id	user_id
592	500
593	501

courier_actions as t2

order_id	courier_id
592	148
594	57

→

order_id	user_id	order_id1	courier_id
592	500	592	148

(FULL) OUTER JOIN – гарантирует, что данные по колонке key из обеих табл. окажутся в финальной табл. (те все данные)

- симметричен – не важно, в каком порядке записаны табл. (какая слева, а какая справа)
- где сможет приджоинить – то приджоинит, где нет – оставит как в исходных табл. (но все равно выведет в финальную)
- алгоритм: 1) выполн. INNER JOIN; 2) добавл. строки из обеих табл., кот. не соотв. усл. ON (те все оставшиеся строки)

```
SELECT*
FROM t1 FULL JOIN t2 ON t1.order_id = t2.order_id
```

user_actions as t1

order_id	user_id
592	500
593	501

courier_actions as t2

order_id	courier_id
592	148
594	57

→

order_id	user_id	order_id1	courier_id
592	500	592	148
593	501		
		594	57

CROSS JOIN – просто декартово произведение 2ух таблиц (1ый этап всех ост. джоинов) (не пересечение, а все возм. комбинации)

- напр. вечеринка, где 6 человек (3м, 3ж) и все хотят потанцевать со всеми. CROSS JOIN выдаст все возможные комбинации
- не нужно указывать условие соединения ON
- **общий вид:**

```
SELECT*
FROM table1
    CROSS JOIN table2
```

или

```
SELECT*
FROM table1, table2
```

SELF JOIN – объединение таблицы с самой собой.

- присоединение происходит не через SELF JOIN, а с помощью любого др. джоина (выше)
- у таблицы должны быть алиасы

```
SELECT *
FROM
    table AS t1
LEFT JOIN
    table AS t2
```

- может быть полезно для:
  - из имеющихся данных создать пары А–А, А–Б, А–В..
  - сравнить табл. саму с собой

Какой выбрать JOIN?

- иногда данные указаны неверно (в одной из табл. может быть указано не все), поэтому, лучше джоинить к той таблице, кот. полнее (напр. указано 60тыс. order_id, а в другой 59тыс. –> джоиним к первой)
- ориентируемся на то, какую инф–ию нужно гарантировано сохранить

Возможные ошибки:

- если мы достаем в SELECT key, кот. был в JOIN, то нужно указать из какой конкретно таблицы он нужен:

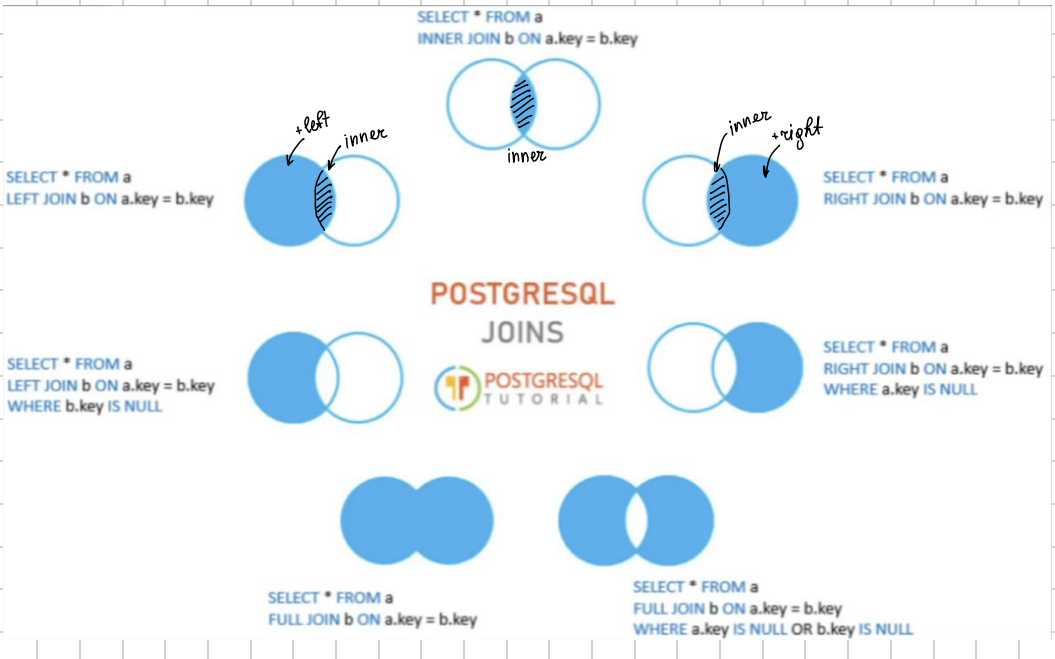
✗ выдаст ошибку:

```
SELECT user_id, order_id, birth_date
FROM user_actions LEFT JOIN users ON user_actions.user_id = users.user_id
```

✓

```
SELECT user_actions.user_id, order_id, birth_date
```

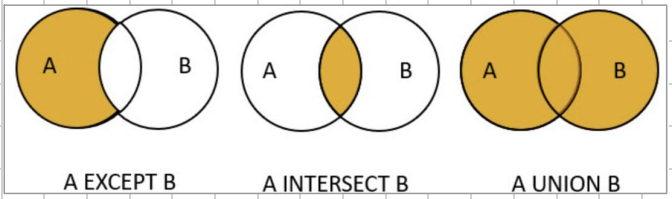
- «пересечение» корректно только для табл. с уникальными (неповторяющимися) key (тк не учитывает, что кол–во строк может увеличиваться, см выше в LEFT JOIN)



Операции с множествами

• исп. чтобы проверить результат, полученный после JOIN
• комбинируют результаты неск. запросов друг с другом –> выдают один общий результат (БД просто отбирает строки, удовл. операции и добавляет их в финальную табл.)
• джоины – объединяют (и модифицируют финальную табл.), а множества – комбинируют
• **общий вид:**

```
SELECT column
FROM table1
[операция с множ.]
SELECT column
FROM table 2
```



EXCEPT – возвращает все записи, кот. есть в 1ом запросе, но отсутствуют во 2ом (разница множеств)

INTERSECT – возвращает все записи, кот. есть в обоих запросах, те и в 1ом и 2ом (пересечение множеств)

UNION – возвращает вообще все записи, те объединяет записи из 2ух запросов в 1 (объединение множеств)

Чтобы работало, надо:

1) в каждом запросе кол–во столбцов в SELECT должно быть одинаковым
2) типы данных в столбцах должны быть совместимы

Оконные функции

Оконные функции – кот. обрабатывают строки наборами (окна/партиции) и записывают результат в отдельном столбце.

- сохраняет исходную структуру табл.
- возвращают ровно то кол-во записей, кот. получили на вход (в отличие от группировки/агрегации)
- используются только! в SELECT и ORDER BY
- в PostgreSql результат оконки будет в формате DECIMAL (не важно в каком формате исходное знач.)
- если нужно отфильтровать/сгруппировать строки после вычисления оконных функций –> исп. подзапрос:

```
FROM (
    SELECT
        user_id,
        date,
        price,
        SUM (price) OVER (PARTITION BY user_id ORDER BY date) AS sum
    FROM table) AS t1
```

- нельзя исп. оконные ф-ии вместе с GROUP BY. Только через подзапросы

√
```
SELECT
    SUM() OVER()..
FROM
    SELECT
        column1
    FROM table
    GROUP BY..
```

✗
```
SELECT
    SUM() OVER()..
FROM table
GROUP BY ...
```

Инструкции при создании окна (они необязательные):

- окна определяются в OVER
- **общий вид:**

```
SELECT OVER (
    PARTITION BY column1, column2, ...
    ORDER BY column 3 ...
    ROWS/RANGE BETWEEN ...)
FROM table
```

0) **пустой OVER()** – исп., если нужно работать со всем набором данных (табл.). Тогда партицией будет вся табл. и сумма будет считаться по всей

```
SELECT
    user_id,
    date,
    price,
    SUM(price) OVER( ) AS sum
```

1) **PARTITION BY** – исп., если нужно группировать данные. Опред. столбец, по кот. данные будут делиться на группы. Опред. партиции внутри окна (по типу GROUP BY)

```
SELECT
    user_id,
    date,
    price,
    SUM(price) OVER(PARTITION BY user_id) AS sum
```

- если PARTITION BY нет, то окно = партиция = вся табл.
- если PARTITION нет и есть ORDER BY, то окно будет вкл. все строки начиная с самой первой и заканчивая текущей (вкл. ее), в том порядке, кот. задает сортировка

2) **ORDER BY ASC/DESC** – исп., если важен порядок. Опред. столбец, по кот. знач. внутри окна будут сортироваться при обработке. Указывается сортировка записей внутри партиций

```
SELECT
    user_id,
    date,
    price,
    SUM(price) OVER(PARTITION BY user_id ORDER BY date) AS sum
```

3) **ROWS/RANGE BETWEEN** – дополнительно задают границы окна и ограничивают диапазон работы функции внутри партиции. (начало рамки, конец рамки)

```
SELECT
    user_id,
    date,
    price,
    SUM(price) OVER(
                PARTITION BY user_id
                ORDER BY date
                ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS sum
```

1) ROWS – начало и конец рамки опред. строками относительно текущей строки
2) RANGE – начало и конец рамки задаются разницей значений в столбце из ORDER BY
   - обязательно указание только 1го столбца в инструкции ORDER BY

Способы задать начало и конец окна (рамки):

1. **UNBOUNDED PRECEDING** – указывает, что рамка начинается с 1ой строки партиции
2. **значение PRECEDING** – указывает, что рамка начинается со сдвигом на заданное число строк относительно тек. строки
3. **CURRENT ROW** – указывает, что рамка начинается/заканчивается на тек. строке
4. **значение FOLLOWING** – указывает, что рамка заканчивается со сдвигом на заданное число строк относительно тек. строки
5. **UNBOUNDED FOLLOWING** – указывает, что рамка заканчивается на последней строке партиции

- знач. PRECEDING и знач. FOLLOWING – только в режиме ROW
- рамка всегда начинается с начала и заканчивается концом
- если конец рамки не указан – подразумевается CURRENT ROW
- **рамка по умолчанию:** RANGE UNBOUNDED PRECEDING, что равносильно:
- **расширенное определение рамки:** RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- если в ORDER BY есть date с типом DATE, то рамка: RANGE BETWEEN '3 days' PRECEDING AND '3 days' FOLLOWING – 3 дня перед и 3 дня после тек. даты, включая ее

пример:

```
SELECT SUM(column3) OVER (PARTITION BY column1
                          ORDER BY column2
                          ROWS BETWEEN UNBOUNDED PRECEDING AND 3 FOLLOWING) AS sum
```

- рамку полезно указывать при расчете скользящее среднее*

*Скользящее среднее – показатель, кот. вычисляется в каждой точке временного ряда как среднее знач. за N предыдущих периодов (дней/мес/..). Скользящее ср. как бы перемещается по временному ряду и каждый раз учитывает фикс. кол–во значений.

Партиция vs Окно

Партиция – подгруппа строк, по кот. происходит разделение данных перед применением оконки.
- (PARTITION BY column) – разделяет данные на группы. Оконка примен. отдельно к каждой партиции (как к отдельной мини табл.)
- если PARTITION BY нет, то окно = партиция = вся табл.

Окно – строки, к кот. применяется оконная ф–ия (кот. исп. для вычисления оконной функции).
- набор строк (те не ограничен колонкой)
- может вкл. строки из нескольких партиций
- (ORDER BY date) – опред. как строки внутри каждой партиции будут отсортированы

Например:

SUM(price) OVER(PARTITION BY user_id ORDER BY date) AS cumulative_sum

user_id	date	price	cumul_sum
1	2025-01-01	10	10 (10+0)
1	2025-01-02	20	30 (10+20)
1	2025-01-03	30	60 (30+30)
2	2025-01-01	15	15
2	2025-01-02	25	40 (25+15)

партиция N.1

партиция N.2

WINDOW

- Окна можно определять через WINDOW, а затем вызывать по алиасу в SELECT:

```
SELECT SUM(column) OVER w AS sum
FROM table
WHERE ...
GROUP BY ...
HAVING ...
WINDOW w AS (
    PARTITION BY ...
    ORDER BY ...
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
ORDER BY ...
LIMIT ...
```

CASE с оконными ф–ями

```
SELECT
    CASE
        WHEN  SUM(column) OVER(...) > 10 THEN 'many'
        WHEN  SUM(column) OVER(...) < 10 THEN 'a_lot'
        ELSE 'a_little'
    END AS ...
```

FILTER с оконными ф–ями

- в окно попадут только те строки, для которых фильтр=истина
- только для агрегирующих оконных ф–ий
- **общий вид:** SELECT agg_function(column) FILTER (WHERE condition) OVER(...)
- пример: SELECT SUM(column1) FILTER (WHERE column2 > 100) OVER (PARTITION BY column3 ORDER BY column4)

В паре с оконными функциями могут использоваться:

1) **Агрегирующие ф–ии:**

• **SUM**, **AVG**, **MAX**, **MIN**, **COUNT**
• внутри окна к ним может применяться ORDER BY
SELECT SUM(column) OVER (PARTITION BY… ORDER BY… ROWS/RANGE BETWEEN…) AS sum

2) **Ранжирующие ф–ии:**

• **ROW_NUMBER** – простая нумерация (1, 2, 3, 4, 5)
• в скобках никогда ничего не указывается. Только в OVER()
• почти всегда нужен ORDER BY
• пример:
ROW_NUMBER() OVER() – почти не исп., тк нет смысла в нумерации без порядка
ROW_NUMBER () OVER(ORDER BY time) – нумерует все строки по возрастанию времени
ROW_NUMBER() OVER(PARTITION BY user_id ORDER BY time) – пронумерует строки отдельно для каждого пользователя

• **RANK** – нумерация с учетом повт. значений с пропуском рангов (1, 2, 2, 4, 5)
SELECT RANK() OVER (PARTITION BY… ORDER BY… ROWS/RANGE BETWEEN…) AS rank

• **DENSE_RANK** – нумерация с учетом повт. значений без пропуска рангов (1, 2, 2, 3, 4)
SELECT DENSE_RANK() OVER (PARTITION BY… ORDER BY… ROWS/RANGE BETWEEN…) AS dense_rank

3) **Ф–ии смещения:**

• **LAG** (назад), **LEAD** (вперед) – значение предыдущей или след. строки
  – 1й аргумент – column
  – 2й аргумент – то, на какое кол–во строк производить смещ. (если не указать, по умолч. = 1)

• **FIRST_VALUE**, **LAST_VALUE** – первое или последнее знач. в окне

*для 2), 3) нужно указывать ORDER BY
*результат нарастающим итогом могут дать только SUM, COUNT, AVG, CUME_DIST + сортировка по ORDER BY (по полям накопл.)
*результат может выглядеть как накопительный итог, если PARTITION нет и есть ORDER BY – окно будет вкл. все строки начиная с самой первой и заканчивая текущей (вкл. ее), в том порядке, кот. задает сортировка

Метрики

ROI (return on investment)

– approximate measure of an investment's profitability.
• helps assess the potential return of investments on a project (stock/business..)
• disadvantage – it doesn't account for how long an investment is held

Can be used to:
– measure the profitability of stock shares
– decide whether to purchase a business
– evaluate the success of a real estate transaction

$$ROI = \frac{\text{Net Return on Investment}}{\text{Cost of Investment}} \cdot 100\%$$

$$ROI = \frac{FVI - IVI}{\text{Cost of Investment}} \cdot 100\%$$

FVI – final value of investment
IVI – initial value of investment

$$\text{Annualised ROI} = [(1+ROI)^{1/n} - 1] \cdot 100$$

n – number of years investment is held

1. bought 1000 shares, $10 each. Sold for $12.50 1 year later. They earned dividends of $500 over the year. Investor spent a total of $125 on commissions (buy and sell).

1.1 $ROI = \frac{(12.50 - 10) \cdot 1000 + 500 - 125}{10 \cdot 1000} \cdot 100 = 28.75\%$

1.2 ROI = Capital gains % – Commission % + Dividend yield
Capital gains = (2500 : 10000) · 100 = 25%
Commissions = (125 : 10000) · 100 = 1.25 %
Dividend yield = (500 : 10000) · 100 = 5%
ROI = 25 – 1.25 + 5 = 28.75 %

В рекламе:
вложено в рекламу 100 ₽
с рекламы продали товаров на 220 ₽
$ROI = \frac{220 - 100}{100} \cdot 100\% = 120\%$
=> на каждый вложенный рубль получили 1.2 ₽ дохода

LTV / CLV ((consumer) lifetime value)

– пожизненная ценность клиента; сколько денег приносит клиент от первой до последней покупки.
• чем больше тем лучше, тк
• ужерживать новых гораздо легче (выгоднее) чем привлечь старых
• помогает выявить самых прибыльных клиентов, работать судержанием, рассчитывать бюджеты, окупаемость, эффективность..
• помогает рассчитать будущий cash flow и сколько новых нужно привлечь для желаемой прибыли
• помогает Со установить «потолок» того, сколько можно тратить на привлечение
• показывает насколько прибыльными были клиенты в прошлом (конкретно этот или max)
• рассчитывается только для Со с one–time purchase revenue model

CPL (cost per lead) — стоимость одного лида
CPO (cost per order) — стоимость одного заказа
CPS (cost per sale) – стоимость одной продажи

Подсчет:
• считается не по выручке, а по прибыли
• LTV > CAC в 3 раза – с т р е м и м с я к этому
• LTV нормальный у всех разный. зависит от бизнеса

LTV = доход за весь период жизни клиента – затраты на его привлечение и удержание
LTV = доход за период / кол–во купивших в этот период
LTV = время сотрудничества человека с брендом * средний доход от него
LTV = средний доход от человека * период сотрудничества * кол–во повторных заказов

$$LTV = \frac{ARPA \cdot \text{Gross Margin}}{\text{Churn rate}}$$

Gross margin % – profit remaining after subtracting the direct costs of the service (e.g. customer service).
ARPA (average revenue per account) – total revenue over a period / total number of active customer accounts during the same period.
Churn rate – pace at which Co expects to lose revenue caused by the loos of customers. Is a revenue attributable to existing customers that are no longer expected to remain customers.
• коэффициент оттока клиентов
• кол–во тех кто больше не взаимодействует с Со (отменили подписку/удалили акк..)
• чем старше компания тем меньше показатель (10% у стартапов, 1% у крупных)

$$\text{Churn rate} = \frac{\text{кол-во ушедших клиентов}}{\text{всего кол-во клиентов}}$$

$$\text{Churn rate}_{\text{за период}} = \frac{\text{кол-во клиентов в начале периода} + \text{новые клиенты} - \text{кол-во клиентов в текущем периоде}}{}$$

ARPU (average revenue per user)

– средний доход от каждого активного покупателя за период.

• норма – сравнить с конкурентами

• главное чтобы не снижался/ Если снижается –> привлекать новых

общий доход за опред. период / число человек, кот. совершили покупку в этот период (пользователи)

ARPPU (average revenue per paying user)

AOV (average order value) – средний чек

выручка за период / общее число заказов за период

За период сервис заработал 100.000 ₽, им воспольз. 500 уникальных пользователей, из ком. 400 сделали 650 заказов

ARPU = 100.000 / 500 = 200

ARPPU = 100.000 / 400 = 250

AOV = 100.000 / 650 ≈ 153,85

CAC (customer acquisition cost)

– стоимость привлечения одного пользователя.

рекламные расходы разделить / число новых пользователей

CRR (customer retention rate)

– доля людей, которые пользуются товарами и услугами компании в конкретный период.

–коэффициент удержания. Показывает долю пользователей, кот. вернулись в приложение/сервис/сайт спустя N дней/недель.. после своего первого входа.

• показатель удержания клиентов

• высокий CRR –> много давнишних клиентов

• хороший уровень: 65% – 85% в зависимости от ниши

((общее число клиентов на конец периода – количество новых за период) / кол–во пользователей в начале периода) * 100%

NPS (net promoter score)

– отношение к компании «готовы ли вы рекомендовать нашу компанию друзьям?».

• проводится опрос, ответ – в баллах по шкале (0 – не буду рекомендовать; 10 – обязательно порекомендую)

• 0–6 – критики

• 7–8 – нейтралы

• 9–10 – промоутеры

• результат [–100; 100], если <0 – плохо, преобладают критики –> поговорить с пользователями и понять что не нравится

$$NPS = \frac{\text{кол-во промоутеров} - \text{кол-во критиков}}{\text{общее кол-во респондентов}} \cdot 100\%$$

MAU (monthly active users). WAU(weekly), DAU(daily)

– кол–во уникальных пользователей за месяц, без учета повторных сессий.

• если 20 человек скачали приложение, но воспользовались 10, то MAU = 10

MAU = кол–во уникальных пользователей за месяц

CAC (customer acquisition cost) – затраты на привлечение 1го покупателя

CAC = затраты на привлечение всех / кол–во привлеченных покупателей

*покупатели это не подписчики, а те, кто совершил покупку

Оптимизация запросов

Оптимизация обычно делается в конце – только при условии, что запрос тормозит

Учитывать при написании, чтобы не пришлось оптимизировать:

- фильтровать данные как можно раньше (напр. перед тем как Join'ить)
- выводить только нужные колонки, лишние убирать
- DISTINCT и GROUP BY сильно нагружают – лучше фильтровать напрямую, чем исп. их
- правильно выбрать JOIN:
  - INNER JOIN самый легкий
  - LEFT JOIN – следующий
  - RIGHT JOIN, FULL JOIN – почти всегда можно переписать
  - *если после LEFT JOIN не понадобились null значения -> заменить на INNER JOIN
- не множить строки в JOIN без необходимости (при связи 1:N). Если необходимо -> агрегировать до JOIN
- убедиться, что JOIN, WHERE, GROUP BY, ORDER BY индексируемы (с помощью EXPLAIN ANALYZE)
- разбивать на CTE. ок до 10 шт., если больше -> EXPLAIN и думать что сократить
- исп. временные табл. или MATERIALIZED CTE

✗
```
SELECT*
FROM orders JOIN users USING(user_id)
WHERE user.city = 'Moscow'
```

✓
```
WITH filtered_users AS(
SELECT user_id
FROM users
WHERE city = 'Moscow')

SELECT*
FROM orders JOIN filtered_users USING(user_id)
```