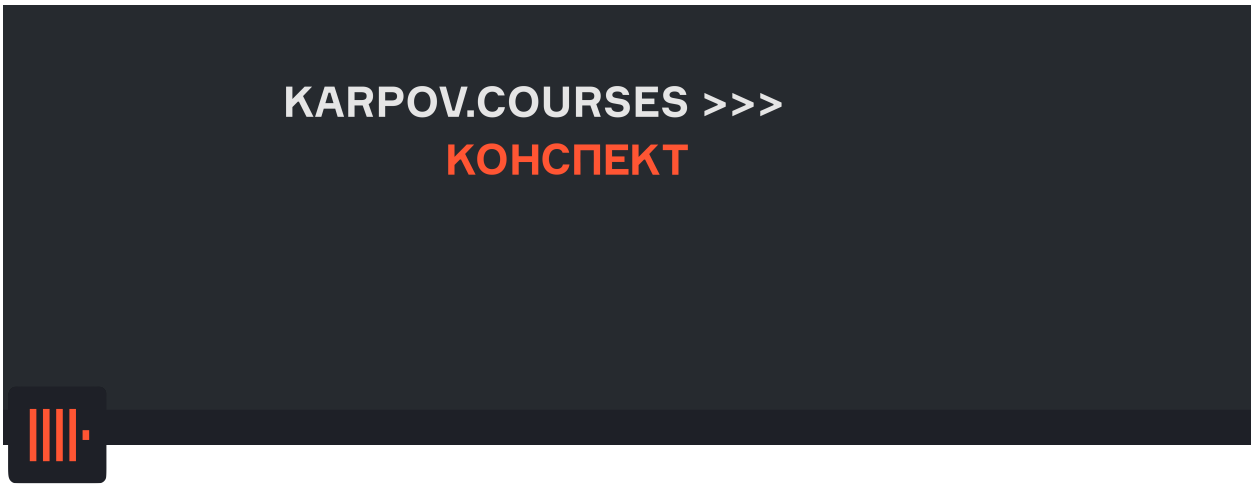

> Конспект > 5 урок > SQL

> Оглавление

> Подзапросы

Встречаются ситуации, когда результат одного запроса нужно использовать как входную таблицу для другого запроса. Например:

- посчитали среднее время от установки до совершение первой покупки для каждого пользователя ← один запрос
- далее хотим усреднить ← второй запрос

```
SELECT
    InstallationDate,
    Source,
    InstallCost / 100 AS cost_fix  -- разделим цену на 100
FROM
    installs
LIMIT 100
```

Теперь используем этот запрос как входную таблицу для другого запроса. Оборачиваем запрос в скобки и записываем его в `FROM` нового запроса. Считаем сумму для каждого источника по месяцам:

```
SELECT
    Month,
    Source,
    SUM(cost_fix) AS sum_cost_fix
FROM
    (SELECT
        InstallationDate,
        Source,
        InstallCost / 100 AS cost_fix
  FROM
        installs
    )
GROUP BY
    toStartOfMonth(CAST(InstallationDate as Date)) AS Month,
    Source
LIMIT 100
```

Самих уровней вложенности может быть сколько угодно. Например, добавим ещё один, где уберем из даты `01`:

```
SELECT
    replaceAll(CAST(Month as String), '-01', '') AS Month_fix,
    Source,
    sum_cost_fix
FROM
    (
    SELECT
        Month,
        Source,
        SUM(cost_fix) AS sum_cost_fix
    FROM
        (SELECT
            InstallationDate,
            Source,
            InstallCost / 100 AS cost_fix
         FROM
            installs
        )
    GROUP BY
        toStartOfMonth(CAST(InstallationDate as Date)) AS Month,
        Source
    )
LIMIT 100
```

В запросах с `JOIN` тоже можно использовать подзапросы:

```
SELECT
    DeviceID,
    UserID,
    Source,
    InstallCost
```

```
FROM (
    SELECT
        DeviceID,
        UserID
    FROM devices
    ) AS l
JOIN (
    SELECT
        Source,
        InstallCost,
        DeviceID
    FROM installs
    ) AS r
ON
    l.DeviceID = r.DeviceID
LIMIT 100
```

> Представления

Представления бывают двух видов:

1. Обычные
2. Материализованные

Обычные представления

Обычные представления – сохранение SQL запроса. Т.е. вы создаете виртуальную таблицу, при вызове которой будет выполняться запрос для её создания. Такие представления на самом деле не содержат данных, а извлекают их из таблицы в момент обращения.

Предположим, что мы очень часто вызываем установки только по iOS. Чтобы каждый раз не писать

```
SELECT * FROM installs WHERE Platform = 'iOS'
```

мы можем создать представление:

```
CREATE VIEW ios_installs AS (
SELECT
    *
FROM
    installs
WHERE
    Platform = 'iOS'
)
```

В результате ClickHouse напишет OK – значит он создал представление. После этого нужно "обновить структуру", и тогда в списке таблиц появится новая – `ios_installs`. Далее представление можно использовать в обычном запросе:

```
SELECT *
FROM ios_installs
LIMIT 100
```

По сути мы сохранили запрос как таблицу: он будет отображаться в списке таблиц как таблица, но на самом деле таблицей не является. Каждый раз, когда вызываем `VIEW`, код запроса выполняется.

Может быть удобно, если вы написали очень большой запрос и планируете часто переиспользовать.

Документация

Материализованные представления

Создаются командой `CREATE MATERIALIZED VIEW`. Также нужно указать движок таблицы, о них поговорим немного позже!

```
CREATE MATERIALIZED VIEW android_installs ENGINE = Log POPULATE AS (
SELECT
    *
FROM
    installs
WHERE
    Platform = 'android'
)
```

В чем отличие материализованного представления от обычного? Материализованное хранит данные на диске, т.е. больше похожа на таблицу. Когда мы вызываем обычное представление, выполняется запрос, а когда материализованное – данные поднимаются с диска.

```
SELECT *
FROM android_installs
LIMIT 100
```

Данные в материализованных представлениях обновляются тогда, когда обновляются те таблицы, по которым они считаются. В данном случае, как только запишутся новые строки в `installs`, материализованное представление обновится и допишет их к себе.

> WITH

`WITH` – используется для объявления параметров. В ClickHouse можно использовать только скалярные выражения, т.е. результат подзапроса должен быть одной строкой.

Допустим мы знаем среднюю цену установки:

```
SELECT
    AVG(InstallCost)
FROM
    installs
-- 69.827832
```

Создаем приближение:

```
-- объявляем параметры
WITH
    (SELECT AVG(InstallCost) FROM installs) AS avg_install_cost,
     sum_rub * 100 AS rub_cent

-- используем
    SELECT
    UserID,
    sum_rub - avg_install_cost AS economy,
    rub_cent
FROM (
    SELECT
        UserID,
        SUM(Rub) AS sum_rub
    FROM
        checks
    GROUP BY
        UserID
    )
LIMIT 100
```

> Команды для работы с таблицами

CREATE TABLE

`CREATE TABLE` – создать таблицу (обязательно указать движок)

```
CREATE TABLE geobase ENGINE = Log AS (
SELECT
  'Moscow' AS City,
  'Russia' AS Country
)
```

```
SELECT *
FROM geobase
```

```
+--------+---------+
|   City | Country |
+--------+---------+
| Moscow |  Russia |
+--------+---------+
```

DROP TABLE

`DROP TABLE` – удалить таблицу.

```
DROP TABLE android_installs
```

INSERT INTO

`INSERT INTO` – запись данных в таблицу. Названия полей и типы данных должны совпадать с исходной таблицей.

```
-- скопируем таблицу с чеками для примера, чтобы не сломать исходную
CREATE TABLE checks_copy ENGINE = Log AS (
SELECT *
FROM checks
)

INSERT INTO checks_copy
    -- немного меняем данные, умножив рубли, добавив 100 дней к дате
SELECT
        Rub * 0.87 AS Rub,
        CAST((CAST(BuyDate AS Date) + 100) AS String) AS BuyDate, -- возвр. к изначальному типу данных String
        UserID * 0.99
FROM checks
```

Проверим, добавились ли данные, должно быть в два раза больше:

```
SELECT COUNT(*) AS rows FROM checks -- 56 088 728
SELECT COUNT(*) AS rows FROM checks_copy -- 112 177 456, все верно!
```

> ALTER

`ALTER` – команда для изменения данных. Для начала, **копируем** таблицу, чтобы не трогать исходные данные:

```
CREATE TABLE listings_copy ENGINE = MergeTree
ORDER BY (id, host_id) SETTINGS index_granularity = 8192 AS (
SELECT *
FROM listings
)
```

ADD COLUMN

`ADD COLUMN` – добавить колонку. Например, `add_column_test` с типом String, вставить после `reviews_per_month`

```
ALTER TABLE listings_copy ADD COLUMN add_column_test String AFTER reviews_per_month
```

MODIFY COLUMN

`MODIFY COLUMN` — изменяет тип столбца

```
ALTER TABLE listings_copy MODIFY COLUMN host_id UInt64
```

DROP COLUMN

`DROP COLUMN` – удалить колонку

```
ALTER TABLE listings_copy DROP COLUMN add_column_test
```

DELETE WHERE

`DELETE WHERE` – удалить строки для которых выполняется указаное условие. Например, все строки, где тип комнаты – 'Private room'.

```
ALTER TABLE listings_copy DELETE WHERE room_type='Private room'
```

Note: не нужно изменять таблицы, на которых основаны задания :)