RIPE NCC
RIPE NETWORK COORDINATION CENTRE

Historical analysis of RIPE Atlas data

ENOG17 | Alex Semenyaka

Goals of this tutorial

- **To share my own real experience with the historical analysis of the RIPE Atlas data**
  - I.e. getting data related to the objects in question when the meta-data (measurement IDs) are unknown
- **To discuss different approaches to do this**
  - And get understanding of their advantages and drawbacks
- **To provide newbies with some DOs and DONTs**

All code snippets provided are written on Python3 but easily could be re-written on any other language

Who can it be useful for?

- For engineers who investigate some network events in the past
  - Especially important if they did not run own measurement in advance
- For researchers studying Internet development phenomena, patterns and trends
- For specialists doing fact checking regarding some happening when the Internet involved

Basic model for this tutorial

- **There are a list of networks**
- **There is a time range in which network events of interest could occur**
  - To simplify the code, we will limit the time range to the **November 1, 2020**
- **We are interested in our getting all measurement results in this time range regarding the networks in our list**
- **The above code is only a PoC (in particular, there are no error checks, timezone is GMT etc)**

This model was chosen as the most typical one.

Approaches outlined in this tutorial may be naturally extended to any other tasks that arising when working with historical data.

What is RIPE Atlas?

RIPE Atlas is the RIPE NCC's main Internet data collection system. It is a global network of devices, called probes and anchors, that actively measure Internet connectivity. Anyone can access this data via Internet traffic maps, streaming data visualisations, and an API. RIPE Atlas users can also perform customised measurements to gain valuable data about their own networks.

Approaches to be discussed

- RIPE Atlas API
- Direct access to the RIPE Atlas storage
- RIPE Atlas data in Google BigQuery

RIPE Atlas API

The most standard way
to do things

What to do

- **Why it was not that straightforward before?**
  - The endpoint https://atlas.ripe.net/api/v2/measurements/ had no parameter to filter out all (or the most of) unnecessary measurements - fixed
  - The number of results can be too high (thus, we can hit ‘20,000 objects limit’)
- **How could this issue be solved?**
  - If we are interested in some IP-addresses, it could be only the IP-addresses either **of the probes** or **of the targets**
  - Therefore, we can split the task into two smaller ones:
    1. Find the probes located in networks in interest, and collect their results
    2. Find the measurements targeted to networks in interest

Measurements from the given probes

- **First of all, the description of all active probes is stored here (daily):** **https://ftp.ripe.net/ripe/atlas/probes/archive/<YYYY>/<mm>/<YYYY><mm><dd>.json.bz2**
  - This file is one big JSON with the information of all probes to the moment it was created
- **There we can pick up the IDs of the probes in our prefixes for the given time interval in the past**
  - Because we want to know the information about the past (not the current situation), we must not use the https://atlas.ripe.net/api/v2/probes/ endpoint

Structure of the description

```
{'meta':    {…},
 'objects': [{'address_v4': '82.95.114.207',
              'address_v6': '2001:983:ba7e:1:220:4aff:fec8:23d7',
              'asn_v4': 3265,
              'asn_v6': 3265,
              'status': 1,
             …},
            …]
```

Code: filter out the probes

```
import bz2
from urllib.request import urlopen, urlretrieve
import json
from netaddr import *

URL = 'https://ftp.ripe.net/ripe/atlas/probes/archive/2020/11/20201101.json.bz2'

networks = [IPNetwork(_) for _ in ('82.209.232.0/24', '37.212.0.0/14')]

with urlopen(URL) as bzstream:
    decoded  = bz2.open(bzstream, 'r')
    allprobes = json.loads(decoded.read())

for probe in allprobes['objects']:
    ipv4 = probe['address_v4']
    if not ipv4:
        continue
    for net in networks:
        if ipv4 in net:
            print(probe['id'])
```

How to get measurements from the given probes

- First, we get the measurement IDs from the https://atlas.ripe.net/api/v2/measurements/
- Then we extract the results for each measurement ID using the endpoint https://atlas.ripe.net/api/v2/measurements/<ID>/results and filter out the relevant for us

Collecting measurement IDs ("from"): code

```
import requests

API_EP    = 'https://atlas.ripe.net/api/v2/measurements/?start_time__lte={}
&stop_time__gt={}&participant_logs_probes={}'
PROBES    = (3596,3569,3986,4473,6878,18921,19445,19968,19975,19977,
            19997,25114,32622,32627,32628,33212,54148,55796,1000444,
            1000446,1000869,1000876,1000878,1001092,1001243,1001244)
start_ts  = '1604188800'           # 01/11/2010
stop_ts   = '1604275200'           # 02/11/2020

msms_ids = set()

for probe in PROBES:
    api_call = API_EP.format(stop_ts, start_ts, str(probe))
    rc       = requests.get(api_call)
    msms     = rc.json()

    for measurement in msms['results']:
        msms_ids.add(measurement['id'])

print(msms_ids)
```

Starts before the end of the time range

Ends after the beginning of the time range

Measurements *towards to* the given prefixes

- There is a parameter of the endpoint https://atlas.ripe.net/api/v2/measurements/ to filter on the target IP address:

target_ip=prefix

- Sometimes can be easier to use another API parameter filtering on ASN:

target_asn=ASN

- Alternatively, RIPE Stat API can be used: https://stat.ripe.net/data-api#atlas-targets

Collecting measurement IDs (“to”): code

```
import requests
from urllib.parse import quote

API_EP   = 'https://atlas.ripe.net/api/v2/measurements/?start_time__lte={}
&stop_time__gt={}&target_ip={}'
PREFIXES = ('194.158.192.0/19', '82.209.192.0/18', '86.57.128.0/17',
            '93.84.0.0/15', '178.120.0.0/13', '37.44.64.0/18',
            '37.45.0.0/16', '37.212.0.0/14', '185.152.136.0/22')
start_ts  = '1004188800'           # 01/11/2010
stop_ts   = '1604275200'           # 02/11/2020

msms_ids = set()

for prefix in PREFIXES:
    api_call = API_EP.format(stop_ts, start_ts, quote(prefix, safe=''))
    rc    = requests.get(api_call)
    msms = rc.json()

    for measurement in msms['results']:
       msms_ids.add(measurement['id'])

print(msms_ids)
```

Getting the results in interest: code

```
import requests

API_EP    = 'https://atlas.ripe.net/api/v2/measurements/{}/results'
MSMS      = (27921412, 26080776, 26080777, 27925515, 27924506, 27924515)

for msm_id in MSMS:
    api_call = API_EP.format(str(msm_id))

    rc   = requests.get(api_call)
    msms = rc.json()

    for result in msms:
        <check result['dst_addr'] and do smth>
```

What to think of — IMPORTANT

- **The number of results can be too high**
  - API results are paged, i.e. after getting of a chunk it is necessary to check if there is more
  - Also, we can hit ‘20,000 objects limit’
- **Network error and issues should be taken in care**
  - Many API requests mean that the failure in the midst of the code could lead to starting from zero
  - Therefore, except with the simplest cases, it is necessary to provide error checking at each step and start the logic of re-requests

References

- **https://atlas.ripe.net/docs/api/v2/manual/**
  - RIPE Atlas Manual
- **https://atlas.ripe.net/docs/api/v2/reference/**
  - RIPE Atlas Reference

RIPE Atlas data storage

Quick-and-dirty solution

Raw Atlas data

- Real Bigdata
- Last results are still publicly available:
  - URL: https://data-store.ripe.net/datasets/atlas-daily-dumps/
  - It keeps all measurement results collected by the RIPE Atlas *during **the last month***

sslcert-2020-09-22T1900.bz2	23-Sep-2020 07:05	18M
sslcert-2020-09-22T2000.bz2	23-Sep-2020 07:04	18M
sslcert-2020-09-22T2100.bz2	23-Sep-2020 07:05	18M
sslcert-2020-09-22T2200.bz2	23-Sep-2020 07:05	17M
sslcert-2020-09-22T2300.bz2	23-Sep-2020 07:05	18M
traceroute-2020-09-22T0000.bz2	23-Sep-2020 06:14	1.3G
traceroute-2020-09-22T0100.bz2	23-Sep-2020 06:15	1.3G
traceroute-2020-09-22T0200.bz2	23-Sep-2020 06:11	1.3G
traceroute-2020-09-22T0300.bz2	23-Sep-2020 06:11	1.3G
traceroute-2020-09-22T0400.bz2	23-Sep-2020 06:15	1.3G
traceroute-2020-09-22T0500.bz2	23-Sep-2020 06:10	1.3G
traceroute-2020-09-22T0600.bz2	23-Sep-2020 06:10	1.3G
traceroute-2020-09-22T0700.bz2	23-Sep-2020 06:12	1.3G
traceroute-2020-09-22T0800.bz2	23-Sep-2020 06:10	1.3G
traceroute-2020-09-22T0900.bz2	23-Sep-2020 06:13	1.3G

Screenshot of the file listing of the Atlas data storage

Files in the storage

- **Each file contains the measurements of the given type made during 1 hour**
  - The special type “connection” describes switching probes online/offline

ping-2020-10-21T1300.bz2	22-Oct-2020 06:46	1.0G
ping-2020-10-21T1400.bz2	22-Oct-2020 06:54	1.0G
ping-2020-10-21T1500.bz2	22-Oct-2020 06:50	1.0G
ping-2020-10-21T1600.bz2	22-Oct-2020 06:49	1.0G
ping-2020-10-21T1700.bz2	22-Oct-2020 06:49	1.0G
ping-2020-10-21T1800.bz2	22-Oct-2020 06:46	1.0G
ping-2020-10-21T1900.bz2	22-Oct-2020 06:46	1.0G
ping-2020-10-21T2000.bz2	22-Oct-2020 06:47	1.0G
ping-2020-10-21T2100.bz2	22-Oct-2020 06:47	1.0G
ping-2020-10-21T2200.bz2	22-Oct-2020 06:47	1.0G
ping-2020-10-21T2300.bz2	22-Oct-2020 06:48	1.0G

- **The name contains the type of measurement and the stamp when the file was created:**
  - <type-of-measurement>-<YYYY>-<mm>-<dd>T<HH>00.bz2

ping-2020-10-21T1800.bz2:

ICMP measurements

Date: 2020-10-21 (October 21, 2020)

Time: 18:00 UTC

What is inside?

- Each file contains all measurements of the given type made during the corresponding hour
- One measurement, one line
- Each measurement is written as a separate JSON object containing all its data

{“fw”:5020,”mver”:"2.2.1","lts":1122859,"dst_name":"92.223.65.18","af":4,"dst_addr":"92.223.65.18","src_addr":"91.240.92.5","proto":"ICMP","ttl":56,"size":64,"result":[{"rtt":36.525277},{"rtt":36.571163},{"rtt":36.602452}],"dup":0,"rcvd":3,"sent":3,"min":36.525277,"max":36.602452,"avg":36.5662973333,"msm_id":25637026,"prb_id":6816,"timestamp":1603306794,"msm_name":"Ping","from":"91.240.92.5","type":"ping","group_id":25637025,"step":240}

{"fw":5020,"mver":"2.2.1","lts":1122859,"dst_name":"2803:4dc0:254::254","af":6,"dst_addr":"2803:4dc0:254::254","src_addr":"2a0a:d880:0:200::5","proto":"ICMP","ttl":49,"size":64,"result":[{"rtt":157.186851},{"rtt":157.098663},{"rtt":157.178395}],"dup":0,"rcvd":3,"sent":3,"min":157.098663,"max":157.186851,"avg":157.1546363333,"msm_id":14395234,"prb_id":6816,"timestamp":1603306794,"msm_name":"Ping","from":"2a0a:d880:0:200::5","type":"ping","group_id":14395233,"step":240}

How do we treat these files?

- **Straightforward (naive) approach:**
  - Read files through bzip2-filter,
  - Parse each line
  - Check if there is something that we need

```
import urllib.request
import bz2
Import json
from netaddr import *

BZFILE  = 'https://data-store.ripe.net/
datasets/atlas-daily-dumps/2020-07-16/
connection-2020-07-16T0000.bz2'
PREFIX = IPNetwork('194.158.192.0/19')

bzstream = urllib.request.urlopen(BZFILE)
decoded = bz2.open(bzstream, 'r')

for ln in decoded:
    msm_data = json.loads(ls)
    if msm_res['dst_addr'] in PREFIX:
        <do something>
```

What can ever go wrong?

- **Files are huge (I mean, HUGE)**
- **Parsing can be really slow**
  - Depending on what you want to extract
  - It can be so slow that the connection can even die
  - Extracting the data from one file can take more than 1 hour

  ➡ In other words: new data in the storage can be accumulating faster than we are treating the old ones

Why does it happen?

- **Is bzip2 using chunks large enough?**
  - Yes
- **Is the json parsing is fast enough?**
  - Yes
- **So where is the bottleneck?**
  - Data checks, for example: **matching IP-addresses (to select those in the prefixes we are researching)**

The solution: regular expression

- **We know the prefixes to search - thus we can search them in the string *before* parsing**
  - False positive will drop the speed but not significantly
- **To do it faster we can use regex, first forming “aligned” prefixes and concatenating them into the regular expression:**
  - Align:

    185.179.80.0/22 ⇒ 185.179.80. 185.179.81. 185.179.82. 185.179.83.
    2a0a:7d80::/31 ⇒ 2a0a:7d80: 2a0a:7d81:
    185.79.16.0/22 ⇒ 185.79.16. 185.79.17. 185.79.18. 185.79.19

  - Join everything, remembering to escape dots:

    (?:185\.179\.80|185\.179\.81|185\.179\.82|185\.179\.83|2a0a:7d80:|2a0a:7d81:|185\.79\.16\.|185\.79\.17\.|185\.79\.18\.|185\.79\.19\.)

The solution: regular expression

- **We can notice that the resulting regex is far from optimal, especially if we deal with hundreds prefixes**
- **Since we do not use any tricky patterns, there is a method to optimise such regex by organising the original prefixes into the Trie structure**
  - Basically, it groups your prefixes by characters
  - For the example above regex from the Trie will be:
    (?:185\.(?:179\.8(?:0\.|1\.|2\.|3\.)|79\.1(?:6\.|7\.|8\.|9\.))|2a0a:7d8(?:0:|1:))
- **We do not need reinvent the wheel, there are the ready-to-use code**
  - Ex.: https://gist.github.com/EricDuminil/8faabc2f3de82b24e5a371b6dc0fd1e0 (from https://stackoverflow.com/questions/42742810/speed-up-millions-of-regex-replacements-in-python-3)

Regular expression: code

```
from urllib.request import urlopen, urlretrieve
import bz2
import json
from netaddr import *
from ReTrie import Trie
import re

def compiled_prefix_re(prefixlist):
    <…>

PREFIXES = ('194.158.192.0/19', '82.209.192.0/18', '86.57.128.0/17',
            '93.84.0.0/15', '178.120.0.0/13', '37.44.64.0/18',
            '37.45.0.0/16', '37.212.0.0/14', '185.152.136.0/22')
URL = 'https://data-store.ripe.net/datasets/atlas-daily-dumps/2020-11-01/http-2020-11-01T2300.bz2'

re_comp  = compiled_prefix_re(PREFIXES)

with urlopen(URL) as bzstream:
    decoded  = bz2.open(bzstream, 'r')
    for bytestr in decoded:
        line = bytestr.decode('utf-8')
        if not re_comp.search(line):
            continue
        msm_data = json.loads(line)
        <check msm_data['from'] and msm_data['dst_addr'] and do stuff>
```

Could be a bit tricky for masks like /17 and shortened IPv6

The solution: regular expression

- **Applicable to other fields as well**
- **Being used for filtering IP-addresses before converting the line to JSON it makes the code much faster**
  - Approximately 100 times faster with the Trie usage
  - Approximately 50 times faster with the straightforward concatenation of prefixes into the regular expression

Google BigQuery

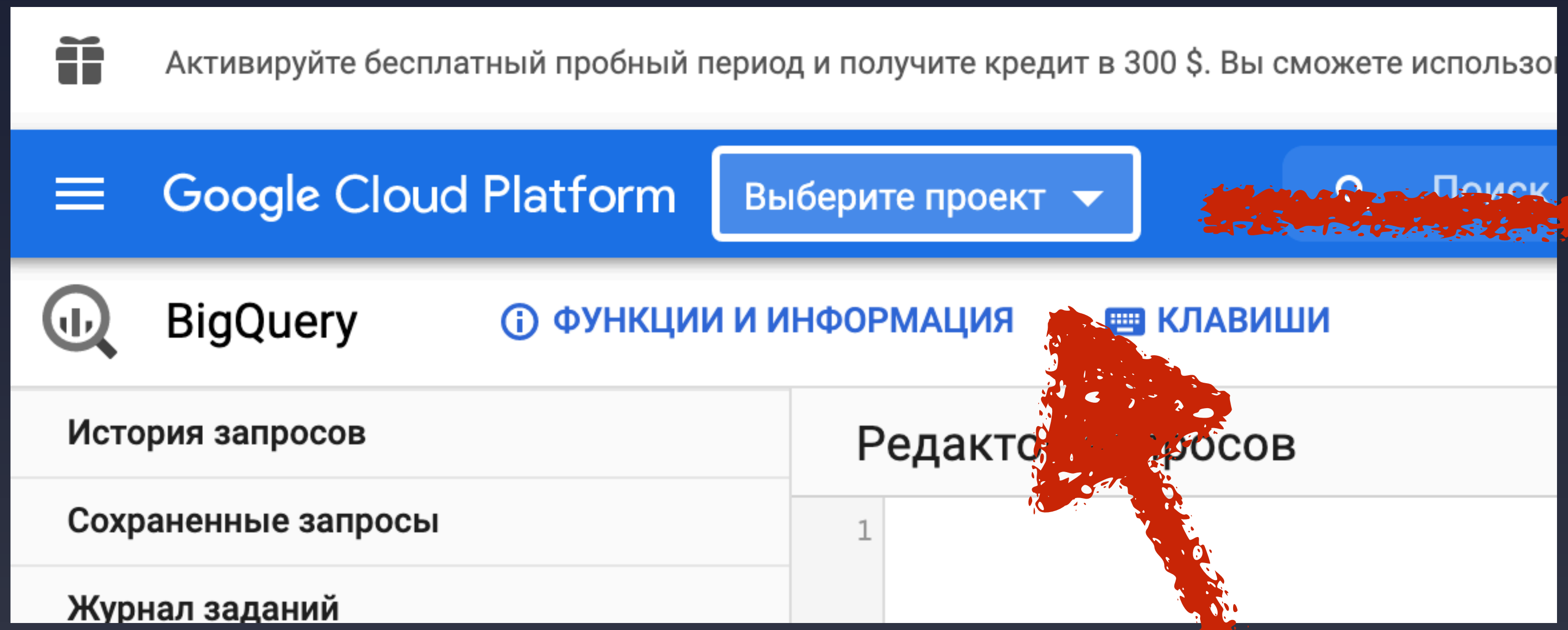
Powerful, but still beta

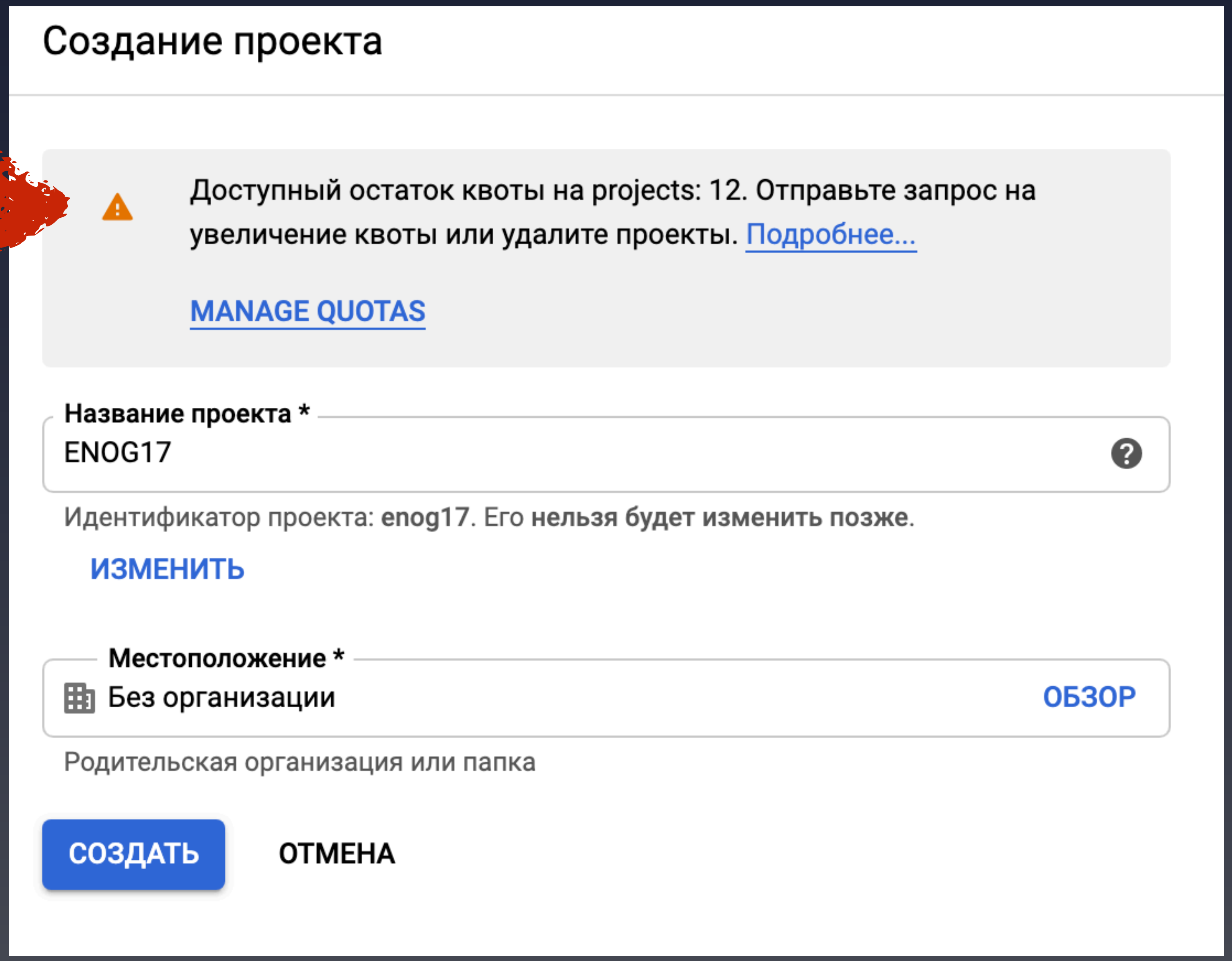
Google BigQuery

- BigQuery is an enterprise data warehouse that solves this problem by enabling super-fast SQL queries using the processing power of Google's infrastructure.
- RIPE Atlas data were uploaded to BigQuery and now are publicly available for BigQuery users
- The manual to start: https://github.com/RIPE-NCC/ripe-atlas-bigquery/blob/main/docs/gettingstarted.md

Step 1: set it up

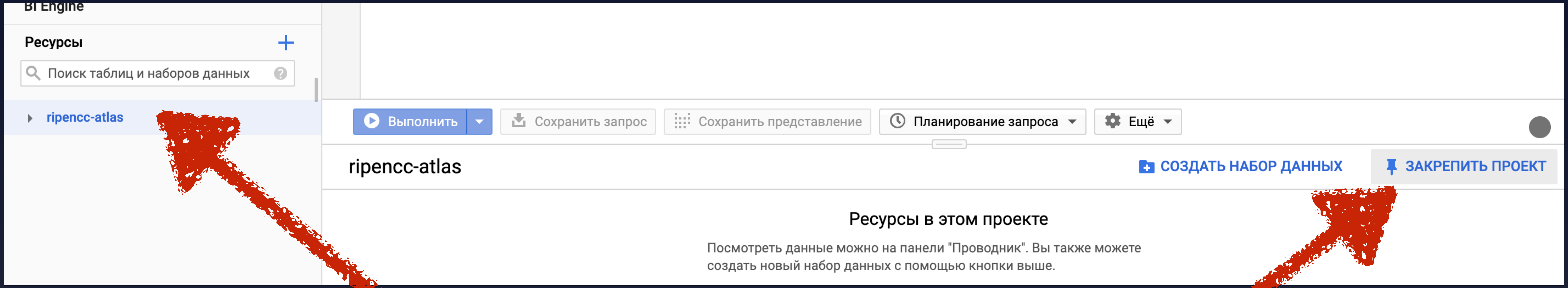
- Make sure you have your account on Google
- Visit https://console.cloud.google.com/bigquery?project=ripencc-atlas





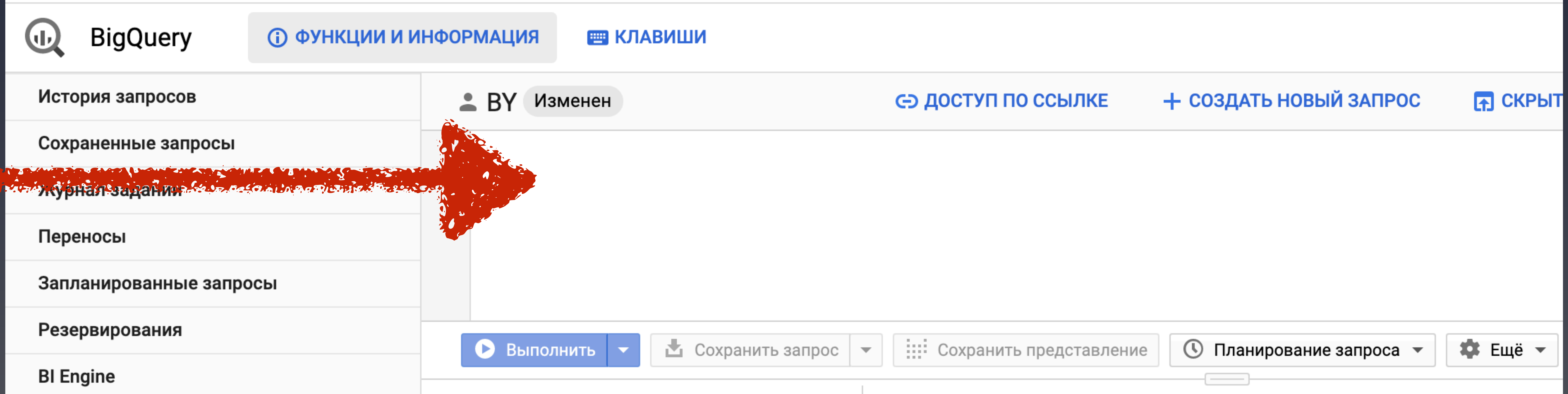
Now, create your own project here

Step 2: prepare the RIPE Atlas data



Select ripencc-atlas and pin it for future

Now we have this field
to play with data



Step 3: a first glance

- **BigQuery uses a SQL-based query language: https://cloud.google.com/bigquery/docs/reference**
- **RIPE Atlas data were uploaded mostly as is**
  - IP addresses has the internal type BYTES to operate with them, so all addresses were converted accordingly
  - start_time has a type TIMESTAMP

Step 3: some howto's

- **Some useful functions:**
  - `REGEXP_EXTRACT(`*`<string>`*`, r'`*`<regex>`*`')`
    - ➡ apply Perl regex to the string and return the match (you can use parenthesis to select what part to return)
  - `NET.IP_FROM_STRING(`*`<string>`*`)`
    - ➡ convert string IP address representation to internal one (BYTES)
  - `NET.IP_TRUNC(<IP-address>, <bits>)`
    - ➡ set lowest bits of the IP-address to 0
  - `SAFE_CAST(<expression> AS <type>)`
    - ➡ cast an expression to the given type
- **Table on fly**
  - `WITH` clause

Step 4: time to play

BY Изменен

ДОСТУП ПО ССЫЛКЕ | + СОЗДАТЬ НОВЫЙ ЗАПРОС | СКРЫТЬ РЕДАКТОР

```
WITH networks AS (
                  SELECT 'by.belpak 194.158.192.0/19' as netstr UNION ALL
                  SELECT 'by.belpak 82.209.192.0/18'            UNION ALL
                  SELECT 'by.belpak 86.57.128.0/17'             UNION ALL
                  SELECT 'by.belpak 93.84.0.0/15'               UNION ALL
                  SELECT 'by.belpak 178.120.0.0/13'             UNION ALL
                  SELECT 'by.belpak 37.44.64.0/18'              UNION ALL
                  SELECT 'by.belpak 37.45.0.0/16'               UNION ALL
                  SELECT 'by.belpak 37.212.0.0/14'              UNION ALL
                  SELECT 'by.belpak 185.152.136.0/22'
                 ),
     netsplit AS (
                  SELECT NET.IP_FROM_STRING(REGEXP_EXTRACT(netstr,r'([0-9a-fA-F\.:]+)/')) AS netaddr,
                         SAFE_CAST(REGEXP_EXTRACT(netstr,r'/([0-9]+)\s*$') AS INT64)      AS netmask   FROM networks
                 )
SELECT * FROM netsplit
```

Выполнить | Сохранить запрос | Сохранить представление | Планирование запроса | Ещё

Результаты запроса

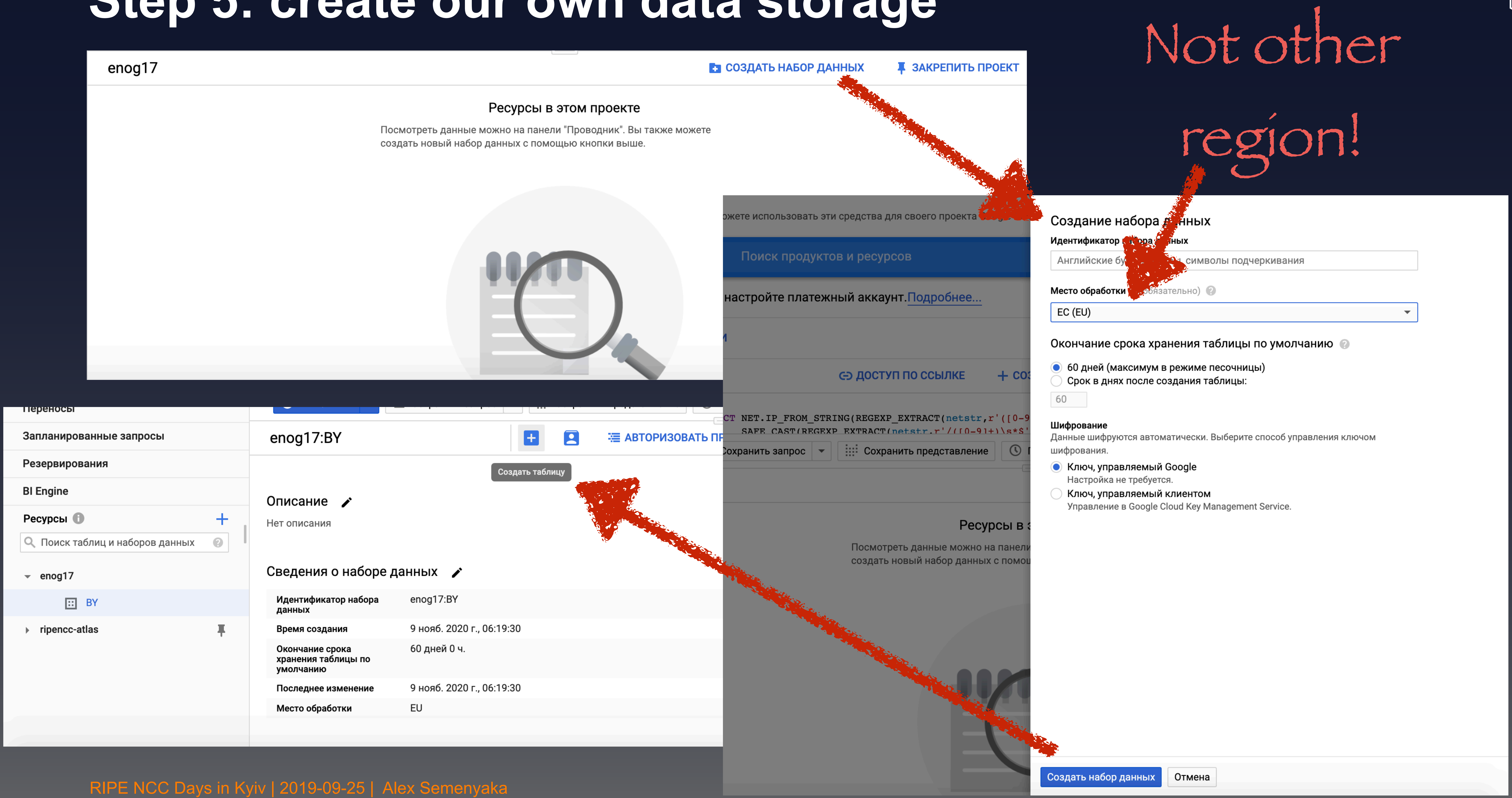
СОХРАНИТЬ РЕЗУЛЬТАТЫ | ПРОСМОТРЕТЬ ДАННЫЕ

Запрос выполнен за 0,3 сек. (обработано 0 Б)

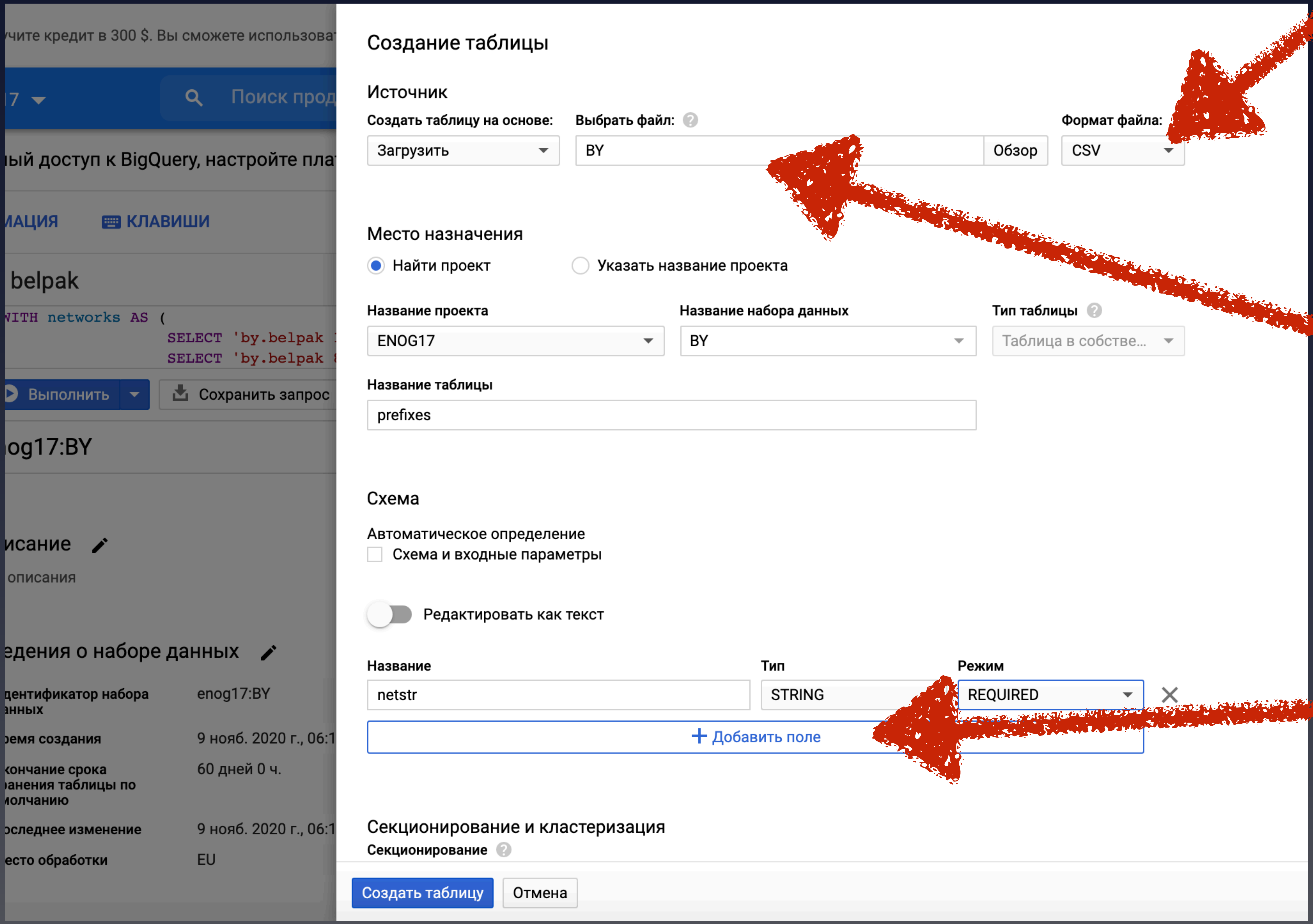
Сведения о задании | Результаты | Данные в формате JSON | Сведения о выполнении

Строка	netaddr	netmask
1	wp7AAA==	19

Step 5: create our own data storage



Step 7: upload our data



File format

Local file

Table structure

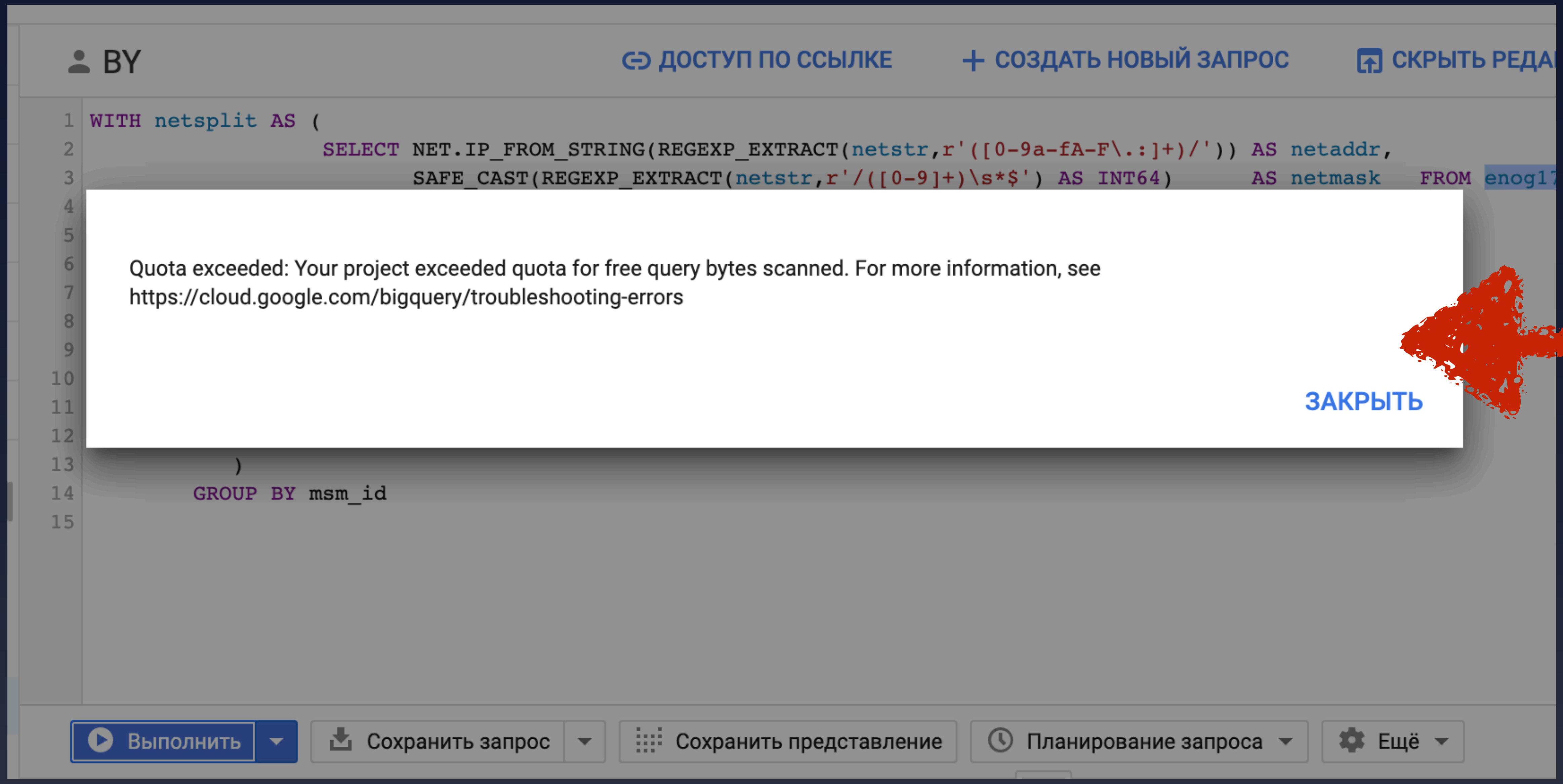
Step 8: use our data together with Atlas

```
WITH netsplit AS (
         SELECT NET.IP_FROM_STRING(REGEXP_EXTRACT(netstr,r'([0-9a-fA-F\.:]+)/')) AS netaddr,
                SAFE_CAST(REGEXP_EXTRACT(netstr,r'/([0-9]+)\s*$') AS INT64)      AS netmask
            FROM enog17.BY.prefixes AS networks
                 )
SELECT msm_id FROM
    netsplit INNER JOIN `ripencc-atlas`.measurements.ping as msmdata
        ON
           ( msmdata.start_time > TIMESTAMP "2020-10-31 00:00:00+00" )
           AND
           (
            ( netsplit.netaddr = NET.IP_TRUNC(msmdata.src_addr_bytes, netsplit.netmask) ) OR
            ( netsplit.netaddr = NET.IP_TRUNC(msmdata.dst_addr_bytes, netsplit.netmask) )
           )
        GROUP BY msm_id
```

Step 9: optimising queries

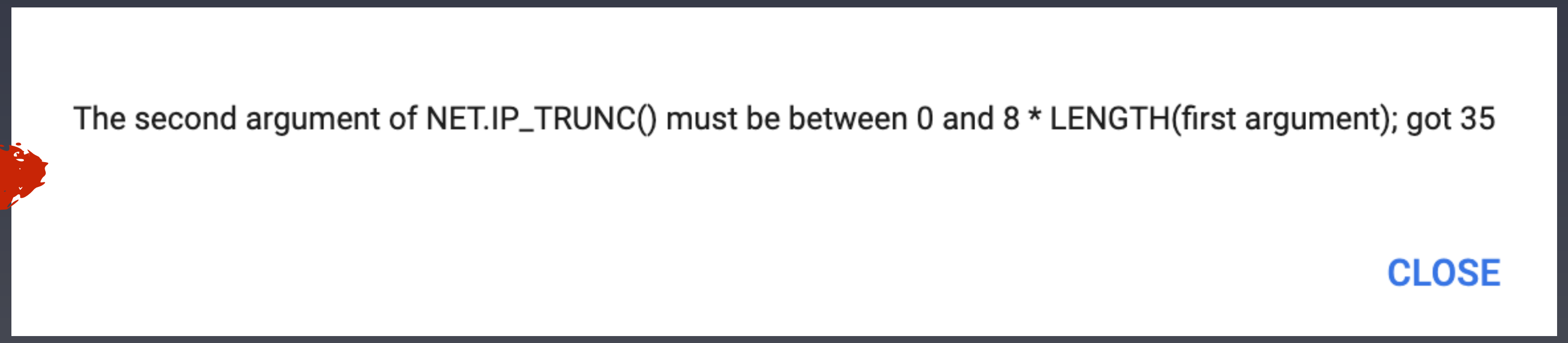
```
INSERT `enog17-295103`.BY.split (name,mask)
    WITH netsplit AS (
           SELECT NET.IP_FROM_STRING(REGEXP_EXTRACT(netstr,r'([0-9a-fA-F\.:]+)/')) AS netaddr,
                 SAFE_CAST(REGEXP_EXTRACT(netstr,r'/([0-9]+)\s*$') AS INT64)       AS netmask
              FROM `enog17-295103`.BY.prefixes AS networks
                 )
    SELECT netaddr, netmask FROM netsplit;

SELECT msm_id FROM
    `enog17-295103`.BY.split AS netsplit INNER JOIN `ripencc-atlas`.measurements.ping as msmdata
        ON
           ( msmdata.start_time > TIMESTAMP "2020-11-05 00:00:00+00" )
           AND
           (
            ( netsplit.name = NET.IP_TRUNC(msmdata.src_addr_bytes, netsplit.mask) ) OR
            ( netsplit.name = NET.IP_TRUNC(msmdata.dst_addr_bytes, netsplit.mask) )
           )
        GROUP BY msm_id
```

Step 10: it happens



Limitations for the free account

Unclear issue with IPv6 addresses



Step 10: first impressions

- Everything is still convenient and ready-to-use
- Real analysis *can* take a while
- Free account may be insufficient for the real work
- Issues with IPv6 addresses
- NCC part: measurement themselves have no data stamp
  - Cannot filter out irrelevant ones

To sum up

API pros and contras

- **Pros**
  - The most mature, robust and universal approach
  - The code is easily reusable in future
- **Contras**
  - A researcher should know programming
  - Complex logic of the code
  - A researches has to deal with all corner cases and internal logic himself
  - It takes a long time to prepare the final code

Storage pros and contras

- **Pros**
  - Easy to start
  - Simple logic, a researcher deals with the measurement results directly
- **Contras**
  - Data available only for the last month
  - Parsing the files in the naive straightforward way can be extremely inefficient

BigQuery pros and contras

- **Pros**
  - Extremely powerful tool
  - Can be easily integrated with other external tools
  - Shared access, easy to use in a team
- **Contras**
  - To use all opportunities of the platform, one should learn a lot
  - Not free
  - From the NCC side:
    - has the beta status
    - measurement timestamps are missing

Questions

asemenyaka@ripe.net