RIPE NCC
RIPE NETWORK COORDINATION CENTRE

CLOUD ATLAS

Stephen Strowes
AIMS 2019
2019-04-16

Introduction

Hadoop at the NCC

Lots of data

- RIPE Atlas generates a lot of measurement data
- In totality, consumes ~66TB (compressed)
- Stored on the NCC's Hadoop cluster(s)

Lots of data

- We need tools that make exploration and analysis of this data easy
- Apache Spark on Hadoop gets us part way there

Running an in-house Hadoop cluster is not easy

- Expenditure: hardware, rack space
- Expenditure: system engineering, maintenance, uptime, patching, user requests, support
- Expenditure: research engineering time

Data Analysis is Exploratory

- Iterative development of an analysis is critical
- Want this to be as tight a loop as possible

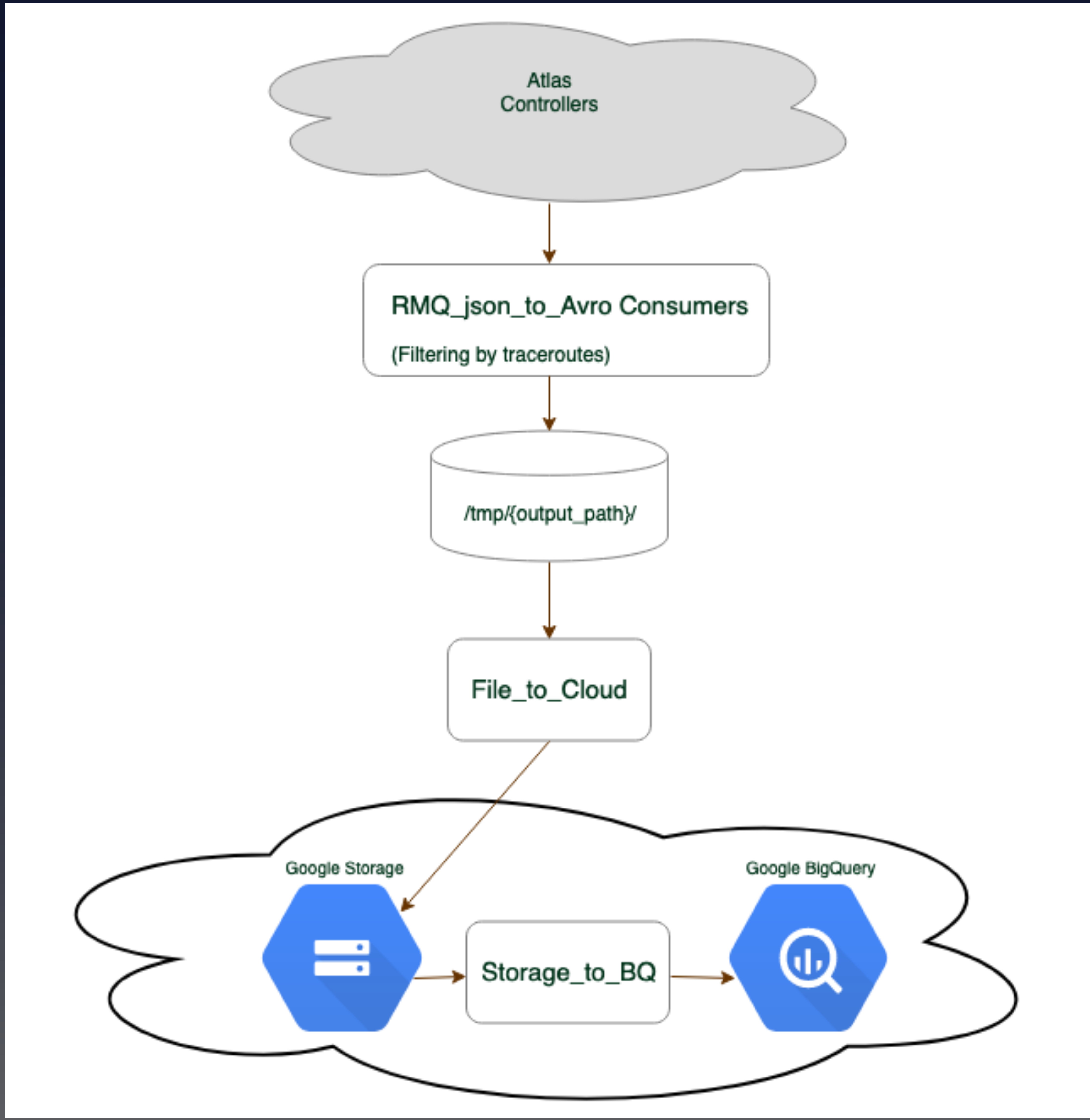
Atlas → Cloud

A prototype

Why the cloud?

- The big three cloud platforms are many years old
  - they reduce expenditure on hardware and time
  - they have SLAs that help keep things running
  - they have all sorts of tooling ready to use (or not use, as we wish)
- We’ve been prototyping against Google Cloud Platform

Prototyping data ingress



Google Cloud Platform

- **Cloud Storage**
  - Avro files dropped in here, to be accessed by BigQuery
- **BigQuery**
  - Data warehouse to store and query massive datasets enabling super-fast SQL queries using the Google infrastructure
  - BigQuery abstracts most everything away

Traceroute data includes nested results


```
{
  "dst_addr": "193.0.19.59",
  "type": "traceroute",
  "dst_name": "193.0.19.59",
  "msm_name": "Traceroute",
  "timestamp": 1551700827,
  "msm_id": 5030,
  "src_addr": "193.0.10.36",
  "prb_id": 6003,
  "from": "193.0.10.36",
  "endtime": 1551700831,
  "result": [
```

```
{
  "hop": 1,
  "result": [
    {
      "rtt": 2.728,
      "ttl": 255,
      "from": "193.0.10.2",
      "size": 28
    },
    {
      "rtt": 2.011,
      "ttl": 255,
      "from": "193.0.10.2",
      "size": 28
    },
    {
      "rtt": 1.628,
      "ttl": 255,
      "from": "193.0.10.2",
      "size": 28
    }
  ]
},
```

```
    {
      "hop": 2,
      "result": [
        {
          "rtt": 107.264,
          "ttl": 62,
          "from": "193.0.19.59",
          "size": 68
        },
        {
          "rtt": 2.122,
          "ttl": 62,
          "from": "193.0.19.59",
          "size": 68
        },
        {
          "rtt": 1.952,
          "ttl": 62,
          "from": "193.0.19.59",
          "size": 68
        }
      ]
    }
  ]
}
```

BigQuery table schema

FIELD	TYPE
IpFrom	STRING
dstAddress	STRING
startTime	TIMESTAMP
endTime	TIMESTAMP
msmId	INTEGER
prbId	INTEGER
groupId	INTEGER
hops	RECORD - REPEATED
hops.hop	INTEGER
hops.resultHops	RECORD - REPEATED
hops.resultHops.rtt	FLOAT
hops.resultHops.from	STRING

BigQuery table schema: example data

IpFrom	dstAddress	startTime	endTime	msmId	prbId	groupId	hop	IpAddHop	rtt
79.127.124.186	193.0.19.109	2019-02-27 04:12:00 UTC	2019-02-27 04:12:14 UTC	2067456	6314	2067456	1	79.127.124.185	1.02
								79.127.124.185	0.785
								79.127.124.185	0.774
							2	172.19.17.65	0.413
								172.19.17.65	0.364
								172.19.17.65	0.385
							3	172.19.17.194	3.765
								172.19.17.194	2.901
								172.19.17.194	2.767

Comparisons

Comparisons

- apples vs. oranges
  - Python with Apache Spark, running on a private Hadoop cluster, vs
  - bigquery running on Google’s own public platform

Example 1

Count IPv6 addrs each probe ran traceroutes to in 1 day

Example 1: pyspark

- Execution time:
    - 16-20 minutes (adhoc queue)
    - 5-6 minutes with a higher priority queue and the cluster isn’t loaded

```
from pyspark import SparkContext
import json

def get_prb_ips( iterator ):
        out = {}
        for d in iterator:
                if d['af'] != 6:
                        continue
                if 'dst_addr' not in d:
                        continue

                prb_id = d['prb_id']
                ip     = d['dst_addr']

                out.setdefault( prb_id, set() )
                out[prb_id].add( ip )

        return out.iteritems()

def collect_sets( a, b ):
        out = a
        for prb_id in b:
                foo = out[prb_id]
                foo = foo.union(b[prb_id])
                out[prb_id] = foo
        return out

def count_ips( a, b ):
        out = a
        for prb_id in b:
                out[prb_id] = len(b[prb_id])
        return out

sc = SparkContext()
reader = sc.sequenceFile("/raw/atlas/day/type=traceroute/2019-04-10.seq/*")

z = reader.map(lambda v: json.loads(v[1]) )
a = z.mapPartitions( get_prb_ips )
b = a.reduceByKey(lambda x,y: x.union(y))

c = b.collect()
d = map( lambda x: (x[0], len(x[1])), c )

for x in d:
        print x
```

Example 1: bigquery

```
select prbId,count(distinct dstAddress)
from   prod.traceroute_atlas_prod_previous_day
where  af = 6
group by prbId
```

- Execution time:
  - 4-5 seconds

Example 2

Find lowest RTT between source and each hop

Example 2: pyspark

- Execution time:
  - ~30 minutes

```
def mp_as_rtts( iterator ):
    '''
    input: raw atlas traceroutes
    '''
    out = {}
    for d in iterator:
        ## array: 0: hops with responses  1: hops without responses
        if 'result' in d:
            if not 'prb_id' in d or not 'dst_addr' in d:
                continue
            for hr in d['result']:
                if 'result' in hr:
                    for h in hr['result']:
                        if 'edst' in h:
                            # doesn't belong in this trace
                            continue
                        if 'from' in h and 'rtt' in h:
                            ip = h['from']
                            key = (d['prb_id'],ip)
                            if (not key in out or out[key] > h['rtt']) and h['rtt'] > 0 and 'late' not in h:
                                out[ key ] = h['rtt']
    return out.iteritems()

trace_path="/raw/atlas/day/type=traceroute/%s.seq" % ( DAY )

# load traceroute
t1 = sc.hadoopFile(trace_path, file_format, key_class, value_class)
t2 = t1.map( lambda v: json.loads(v[1]) )

# finds all RTTs for (src,dst) combinations in a partition
t3 = t2.mapPartitions( mp_as_rtts )
t4 = t3.reduceByKey( min )

t5 = t4.map( lambda x: json.dumps( x )).saveAsTextFile('/user/edominguez/output-ips3')
```

Example 2: bigquery

```
SELECT result.from AS IpAddress, prbId, MIN(result.rtt) AS minRtt
FROM `data-test-194508.prod.traceroute_atlas_prod`, unnest (hops) AS hop, unnest (resultHops) AS result
WHERE startTime >= TIMESTAMP("2019-02-15") and startTime < TIMESTAMP("2019-02-16")
GROUP BY result.from, prbId
```

- Execution time:
  - ~25 seconds

Example 3

Emile's probe similarity work

Example 3: pyspark

- Execution time:
  - ~2 hours

```python
### find similarities between probes, based on the traceroute results
## compare jaccard index per similar measurement, and take 25th,50th,75th percentile

## number of users per day and related
import sys
import numpy
import os
from pyspark.sql import SQLContext, Row
from pyspark.sql import functions as F
import ujson as json
from pyspark.context import SparkContext
from operator import add
import bz2
import radix
import subprocess

sc = SparkContext()
sqlContext = SQLContext(sc)

## http://stackoverflow.com/questions/25193488/how-to-turn-off-info-logging-in-pyspark
def quiet_logs( sc ):
  logger = sc._jvm.org.apache.log4j
  logger.LogManager.getLogger("org"). setLevel( logger.Level.ERROR )
  logger.LogManager.getLogger("akka").setLevel( logger.Level.ERROR )
quiet_logs( sc )

### find all the measurements, and split out the 'system' based ones
DAY=sys.argv[1]
TYPE='traceroute'
## address family
AF=int(sys.argv[2])

r = radix.Radix()
## unconsidered addresses
r.add('10.0.0.0/8')
r.add('172.16.0.0/12')
r.add('192.168.0.0/16')
r.add('100.64.0.0/10')
r.add('fe80::/64')

rtreeBroadcast = sc.broadcast( r )
afBroadcast = sc.broadcast( AF )

def mp_extract_ips( iterator ):
    out = {}
    rtree = rtreeBroadcast.value
    af = afBroadcast.value
    for d in iterator:
        if d['af'] != af:
            continue
        #key = (row['prb_id'],row['dst_addr'])
        #key = "%s|%s" % (row['prb_id'], row['dst_addr'])
        ## array: 0: hops with responses  1: hops without responses
        ips = set()
        if 'result' in d:
            for hr in d['result']:
                if 'hop' in hr:
                    this_hop = hr['hop']
                else:
                    continue
                if 'result' in hr:
                    for h in hr['result']:
                        if 'edst' in h:
                            continue # doesn't belong in this trace!
                        if 'from' in h:
                            ip = h['from']
                            # don't consider dst address and rfc1918 etc.
                            if 'dst_addr' in d and d['dst_addr'] == ip:
                                continue
                            if rtree.search_best( ip ):
                                continue
                            ips.add(ip)
        for ip in ips:
            out.setdefault( d['prb_id'], {} )
            out[ d['prb_id'] ].setdefault( d['msm_id'], set() )
            out[ d['prb_id'] ][ d['msm_id'] ].add( ip )
    return out.iteritems()

## load traces (non cleaned-up)
file_format="org.apache.hadoop.mapred.SequenceFileInputFormat"
key_class="org.apache.hadoop.io.Text"
value_class="org.apache.hadoop.io.Text"
trace_path="/raw/atlas/day/type=traceroute/%s.seq" % ( DAY )

t1 = sc.hadoopFile(trace_path, file_format, key_class, value_class)
t2 = t1.map( lambda v: json.loads(v[1]) )

rdd1 = t2.mapPartitions( mp_extract_ips )

def reduce_nestedset( a, b ):
    out = a
    for msm_id in b:
        out.setdefault( msm_id , set() )
        for ip in b[ msm_id ]:
            out[ msm_id ].add( ip )
    return out

ips_unfilt_rdd = rdd1.reduceByKey( reduce_nestedset )

def fm_weed_out( row ):
    '''
    weed out measurements with less then 2 ips
    weed out probes that only have measurements with less then 2 ips
    '''
    out = {}
    for msm_id,ipset in row[1].iteritems():
        if len( ipset ) > 1:
            out[ msm_id ] = ipset
    if len( out.keys() ) > 0:
        return [ ( row[0], out ) ]
    else:
        return []

## weed out cases where no useful ipset was created
ips_rdd = ips_unfilt_rdd.flatMap( fm_weed_out )

# multinested structure: keys are prb_ids, inside are msm_id: ip_sets
collect = ips_rdd.collectAsMap()

prb_ids = collect.keys()
idx1=0

def compare(data,id1,id2):
    #find the common set of measurements
    msm1 = set( data[id1].keys() )
    msm2 = set( data[id2].keys() )
    msm_set = msm1 & msm2
    metric_per_msm = []
    usable_set_size = 0
    for msm in msm_set:
        ipset1 = data[id1][msm]
        ipset2 = data[id2][msm]
        if len( ipset1 ) < 2 or len( ipset2 ) < 2: # threshold
            continue
        usable_set_size += 1
        ipset_incommon = ipset1 & ipset2
        ipset1_size = len(ipset1)
        ipset2_size = len(ipset2)
        ipset_both_size = len(ipset_incommon)
        ipset1_uniq = ipset1_size - ipset_both_size
        ipset2_uniq = ipset2_size - ipset_both_size
        total_size = ipset1_uniq + ipset2_uniq + ipset_both_size
        if total_size > 0:
            metric_per_msm.append( ipset_both_size * 1.0 / total_size )
    if usable_set_size < 17: # don't output a value if we don't have enough measurements in common
        return
    metric_per_msm = sorted( metric_per_msm )
    metric25 = -1
    metric50 = -1
    metric75 = -1
    (metric25,metric50,metric75) = numpy.percentile( metric_per_msm, [25,50,75] )
    print "%s %s %s %s %s %s %.3f %.3f %.3f" % (
        id1,
        id2,
        len(msm1),
        len(msm2),
        len(msm_set),
        usable_set_size,
        metric25,
        metric50,
        metric75
    )

print "#prb_id1 prb_id2 msm_set_size1 msm_set_size2 msm_set_size_overlap msm_set_size_overlap_usable metric_q1 metric_q2 metric_q3"
for idx1, prb_id1 in enumerate(prb_ids):
    for prb_id2 in prb_ids[idx1+1:]:
        compare(collect,prb_id1,prb_id2)
```

Example 3: bigquery

- Execution time:
  - ~25 minutes

```sql
SELECT msmId, prbId, result.from as ipAdd
    FROM  `prod.traceroute_atlas_prod`, UNNEST(hops) as hop, UNNEST(resultHops) as result
    WHERE  startTime >= TIMESTAMP('{{ds}}') and startTime < TIMESTAMP('{{macros.ds_add(ds,1)}}')
        AND LENGTH (result.fromBytes) = 4
        AND NET.IP_FROM_STRING(dstAddress) <> NET.IP_FROM_STRING(result.from)
        AND result.edst = ''
-- unconsidered addresses  ('10.0.0.0/8') ('172.16.0.0/12')  ('192.168.0.0/16')   ('100.64.0.0/10')   ('fe80::/64')
        AND NET.IP_FROM_STRING(result.from)  not between NET.IP_FROM_STRING("10.0.0.1") and NET.IP_FROM_STRING("10.255.255.254")
        AND NET.IP_FROM_STRING(result.from)  not between  NET.IP_FROM_STRING("172.16.0.1") and NET.IP_FROM_STRING("172.31.255.254")
        AND NET.IP_FROM_STRING(result.from)  not between  NET.IP_FROM_STRING("192.168.0.1") and NET.IP_FROM_STRING("192.168.255.254")
        AND NET.IP_FROM_STRING(result.from)  not between  NET.IP_FROM_STRING("100.64.0.1") and NET.IP_FROM_STRING("100.127.255.254")
    GROUP BY msmId, prbId, ipAdd


WITH
    F1 AS (
        SELECT msmId AS MSM1, prbId AS PRB1, count (IpAdd) AS COUNTIPS1
        FROM `prod_tmp.task1_pair_probes_temporary_table_ipv4` GROUP BY msmid, prbid
    ),
    F2 AS (
        SELECT msmId as MSM2, prbId as PRB2, count(ipAdd) as COUNTIPS2
        FROM `prod_tmp.task1_pair_probes_temporary_table_ipv4` group by msmid, prbid
    )
SELECT MSM1 as msmId, PRB1 as prbId1, prb2 as prbId2, countips1+countips2 as totalIps
    FROM F1, F2
    WHERE MSM1 = msm2 and prb1 < prb2  and countips1 > 1 and countips2 > 1


WITH
    F1 AS (
        SELECT msmId AS MSM, prbId AS PRB, IpAdd AS IP
        FROM `prod_tmp.task1_pair_probes_temporary_table_ipv4`
    ),
    F2 AS (
        SELECT MSM, PRB as PRB1, prbId AS PRB2, IP
        FROM F1, `prod_tmp.task1_pair_probes_temporary_table_ipv4`
        WHERE PRB <> prbId and MSM = msmId and IP = IpAdd and PRB < prbId
    )
SELECT MSM as msmId, PRB1 as prbId1, PRB2 as prbId2, count(IP) as commonIps
    FROM F2
    WHERE PRB1 < PRB2 group by MSM, PRB1, PRB2

SELECT a.prbId1 as prbId1, a.prbId2 as prbId2, ARRAY_AGG(coalesce(commonIps, 0)/(totalIps - coalesce(commonIps, 0))) as distance
    FROM `prod_tmp.task2_pair_probes_count_ips_temporary_table_ipv4`  as a
    LEFT JOIN `prod_tmp.task3_pair_probes_common_ips_ipv4` as b
        ON a.msmId = b.msmId AND a.prbId1 = b.prbId1 AND a.prbId2 = b.prbId2
    GROUP BY a.prbId1, a.prbId2

select prbId1, prbId2, median25, median5, median75
    FROM (
        SELECT prbId1, prbId2,
            percentile_cont (dist, 0.25) over (partition by prbId1, prbId2) as median25,
            percentile_cont(dist, 0.5) over (partition by prbId1, prbId2) as median5,
            percentile_cont(dist, 0.75) over (partition by prbId1, prbId2) as median75
        FROM  `prod_tmp.task4_pair_probes_distance_by_msm_ipv4`, unnest(distance) as dist)
    GROUP BY prbId1, prbId2, median25, median5, median75
```

Takeaways

- But the point is that the abstractions are hidden well by the language *and* processing time is faster
- The end result: **more rapid data analysis**

The Future

The Future

- This is prototype, exploratory work
  - putting other datasets in here, *e.g.*, IPmap data, ping data, peeringdb data
- Project not costed, etc, etc
- But, it looks promising

General Access to Data *and* Tooling?

- Most Atlas data is public, if not always easy to aggregate
- If data is in a commodity cloud system, maybe it can be made more generally accessible
- Give people access to all the data, **and the platform's tooling to operate over that data**, easily
- Get to the science faster?

General Access to Data *and* Tooling?

- Charging models: the NCC provides the data, and researchers pay for compute cycles/network transit they use
- Big vendors support open data initiatives with free storage:
  - https://aws.amazon.com/opendata/
  - https://cloud.google.com/bigquery/public-data/
- This doesn't have to be hosted on Google, but any commodity platform that people are familiar with opens up the measurement data

Questions?

Elena <edominguez@ripe.net>