Table of contents.

Note about statistical significance. **2**

Note about the hadoop distro I used for this task. **3**

New schema. **4**

Partitioning by cityid. 4

Conclusion. 8

Bucketing by cityid. 9

Conclusion. 12

Bucketing by cityid, device, os and browser. 13

Conclusion. 16

Partitioning by cityid and bucketing by device, os and browser. 17

Conclusion. 21

Bucketing the whole table and not just individual columns. 22

Conclusion. 25

Compression. **26**

ORC format with snappy compression. 26

ORC format with zlib compression. 31

Parquet format with snappy compression. 36

Parquet format with zlib compression. 40

Avro format with snappy compression. 44

Avro format with zlib compression. 49

Conclusion. 52

Indexing. **53**

Conclusion. 54

Vectorization. **55**

Tez vs MapReduce. **56**

Conclusion. 59

Final conclusion. **60**

Hive task 2 report.

Note about statistical significance.

When I test for statistical significance, I simply conduct my tests as many times as would be enough to get the value occurring in 20% of the time (or less) at the extremities of my distribution.

Say, my run time results are 78, 79, 79, 79, 80. The values 78 and 80 have 1 in 5 chance to occur, which is 20%, so my run time data is statistically significant. **This is a good metric if the run time data is dense. I am not saying that it will work on sparse data.**

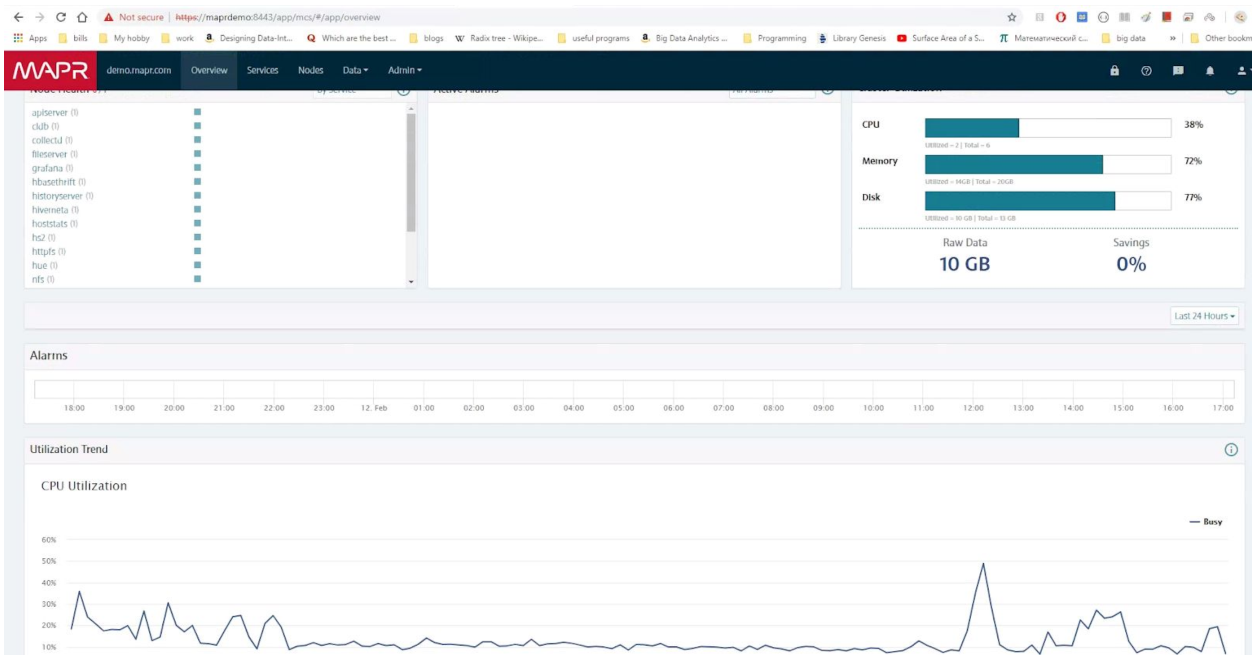
Also, when choosing the best schema, I don't consider the time that it took to write into a new table, because I think that it's a moot point and depends on the use case, but I do include it as a part of this report.

Note about the hadoop distro I used for this task.

I had to resort to using MapR, because neither hortonworks, nor cloudera was able to partition my dataset by the **cityid** column, I spent days upon days on end, trying to get it to work, and just couldn't, I think it's due to both cloudera and hortonworks using HDFS, on the other hand MapR uses MapR-FS which is different and worked out for my use case.

I reached out to hortonworks forums and to stackoverflow regarding this issue here and here and never got a solution.

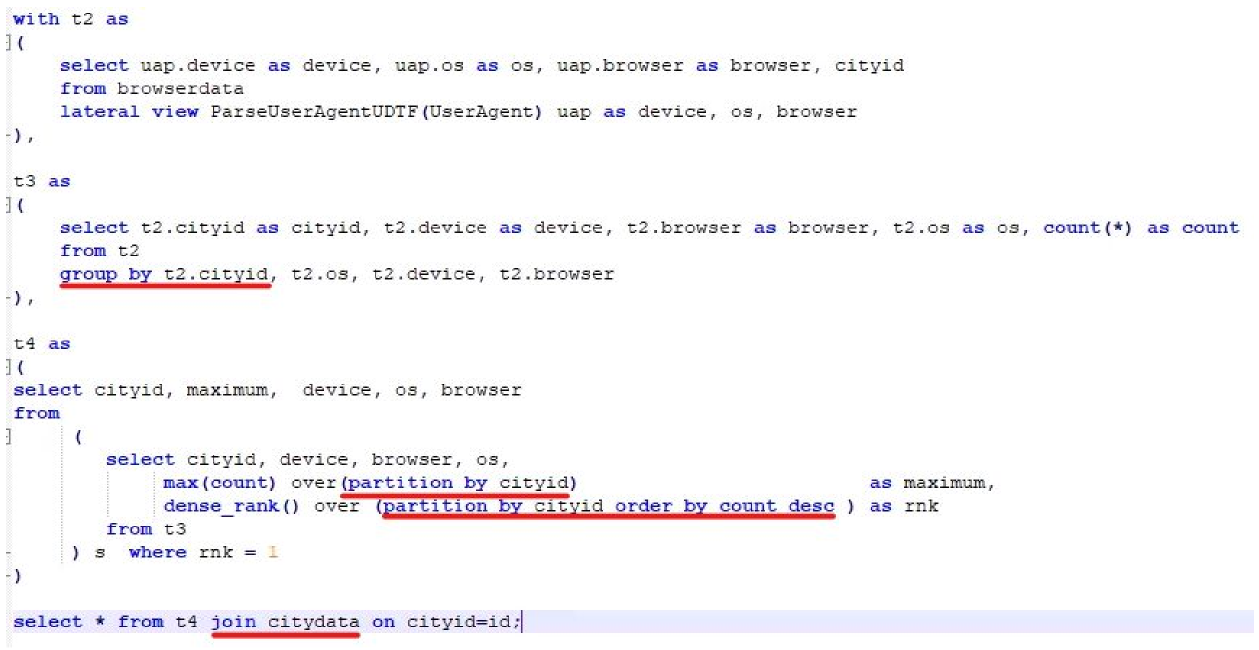
Here's MapR installed and running on my machine:



New schema.

Partitioning by cityid.

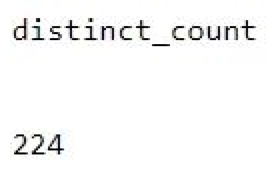
Let’s consider my query:



```
with t2 as
(
    select uap.device as device, uap.os as os, uap.browser as browser, cityid
    from browserdata
    lateral view ParseUserAgentUDTF(UserAgent) uap as device, os, browser
),

t3 as
(
    select t2.cityid as cityid, t2.device as device, t2.browser as browser, t2.os as os, count(*) as count
    from t2
    group by t2.cityid, t2.os, t2.device, t2.browser
),

t4 as
(
select cityid, maximum,  device, os, browser
from
     (
        select cityid, device, browser, os,
             max(count) over(partition by cityid)                         as maximum,
             dense_rank() over (partition by cityid order by count desc ) as rnk
        from t3
     ) s  where rnk = 1
)

select * from t4 join citydata on cityid=id;
```

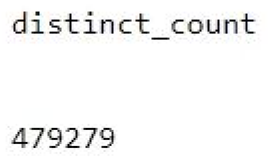
I have underlined the parts that strongly suggest, that I should partition my data by **cityid**, which should be fairly obvious: in every single case, be it **group by**, **partition by** or **join** the query optimizer can simply steer hive in the appropriate direction, make it dive into the necessary partition and grab all the data that it needs instead of scanning through the whole table.

I have also analyzed our data by doing **analyze table browserdata compute statistics for columns cityid;** and after doing **describe formatted browserdata cityid** I get this statistic:



```
distinct_count


224
```

Which even further suggests that **cityid** is a good candidate for being a partition column. Here’s the same statistic for the **useragent** column for comparison:



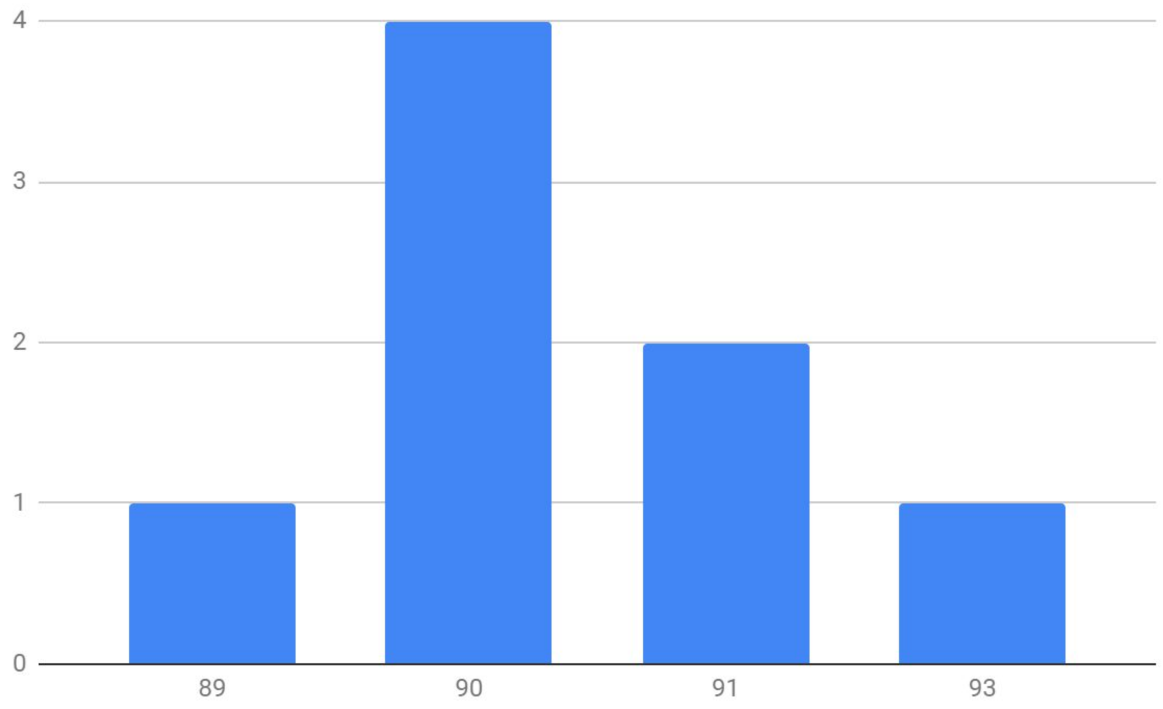
```
distinct_count

479279
```

Before we begin exploring the results of how long our query took, I should also add that ***it took about 915 seconds on average to partition the unpartitioned dataset****.*

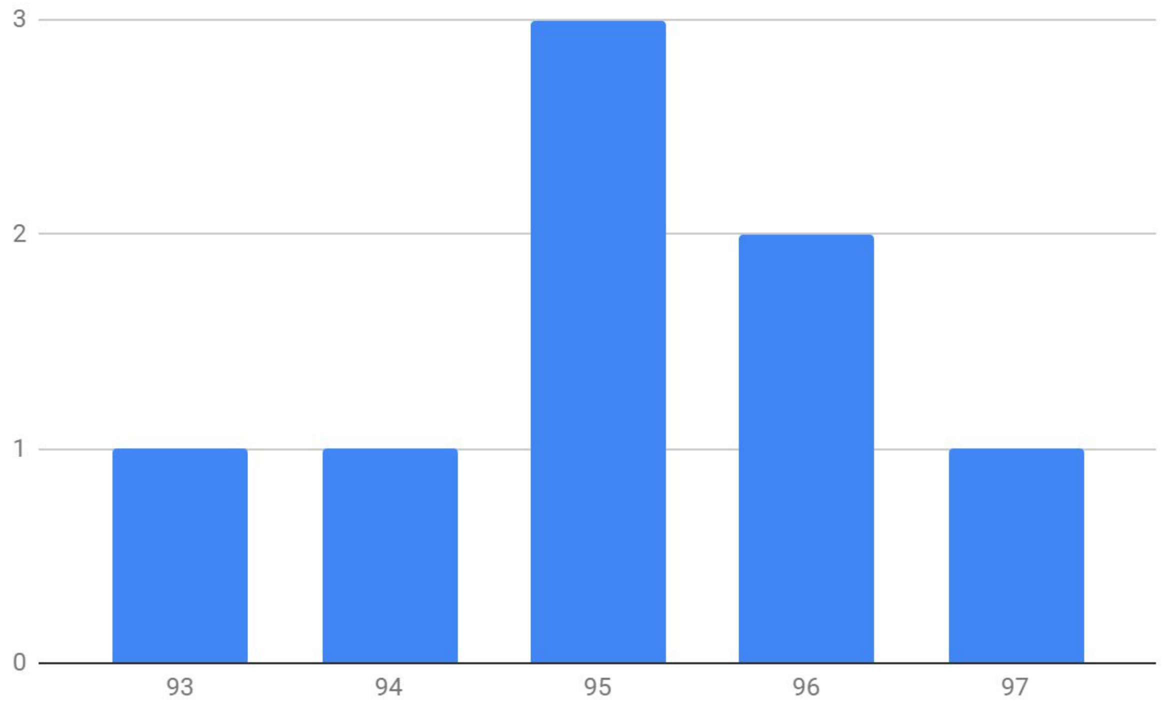
Without further ado, here's the partitioned data results:

Attempt #	Unpartitioned time (in seconds)	Partitioned time (in seconds)
1	89	93
2	90	94
3	93	95
4	91	96
5	90	95
6	90	95
7	91	97
8	90	96
Average	**90,5**	**95,125**

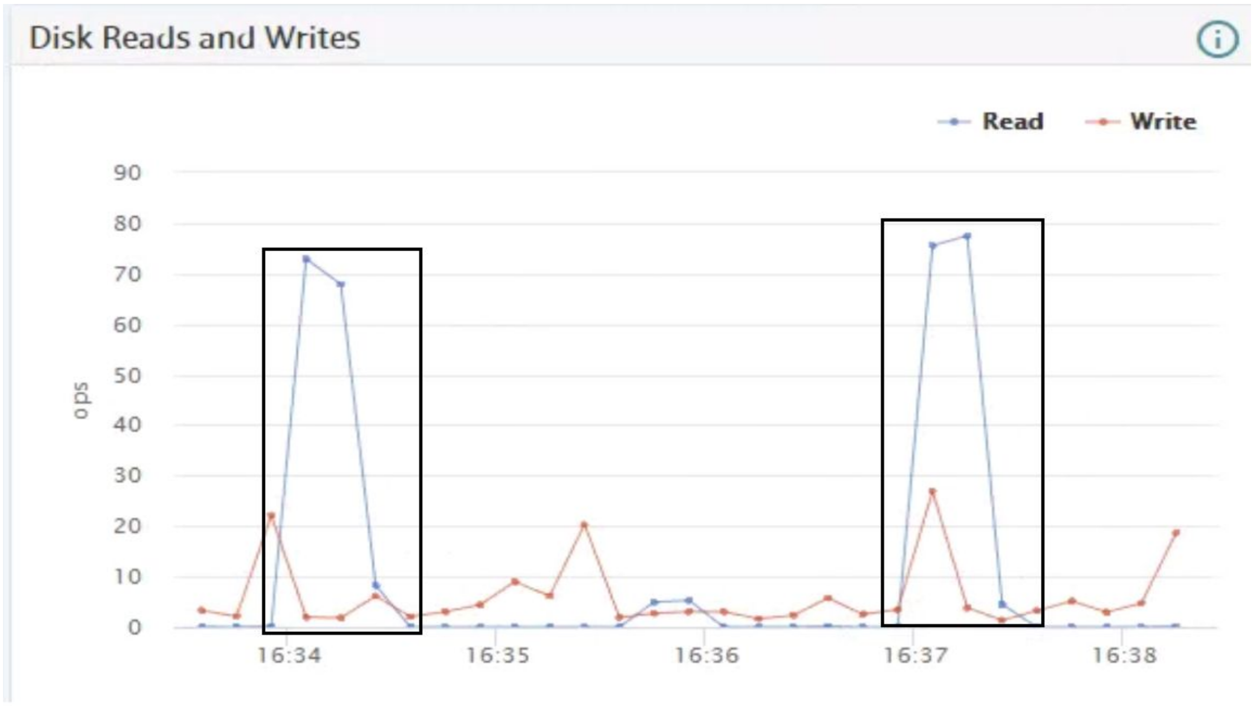
Unpartitioned distribution:



Partitioned distribution:

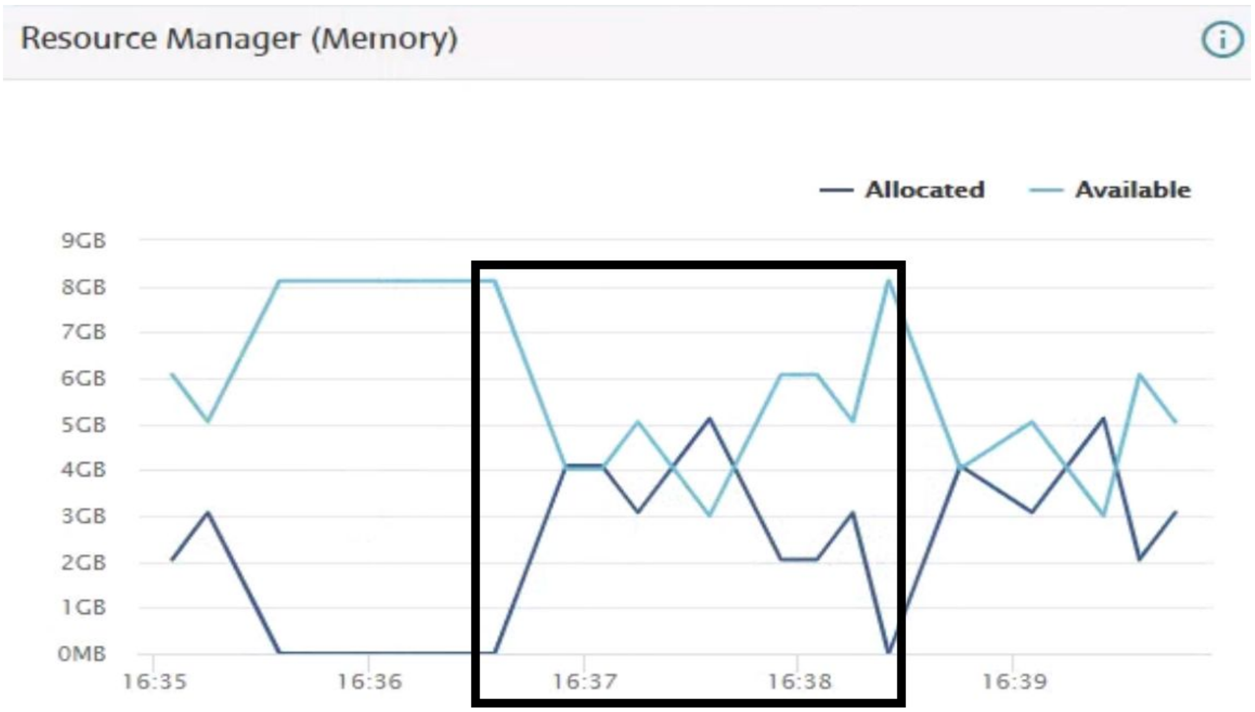


Here's the disk usage after 2 runs of my query on mapreduce:

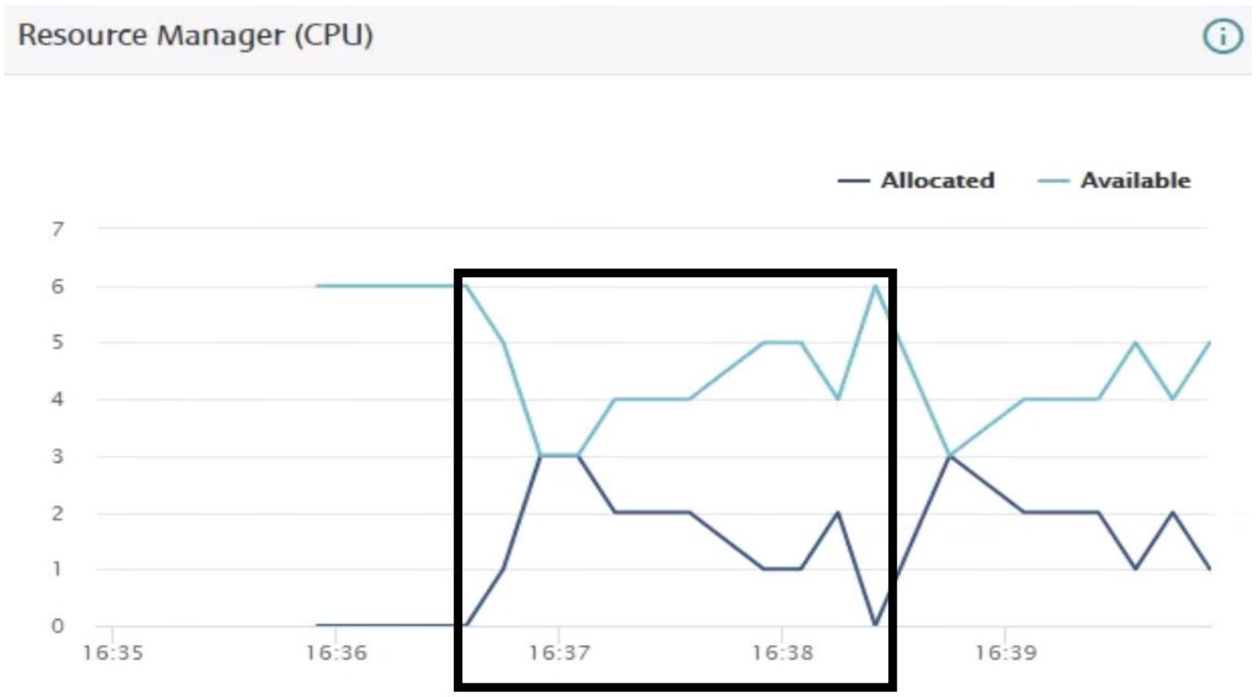


As you can see, we are performing a lot of reads/writes, hence, disk speed is our bottleneck. A typical hard drive can perform a total of 90 operations per second, in our case, hive was performing roughly that number of operations for half of the time that my queries took.

Now, contrast it with the RAM usage (the picture only shows RAM usage of one operation here):



And CPU usage:



It is fairly obvious that disk usage was much more dramatic.

Conclusion.

Alas, my hunch about the need to partition by cityid was wrong, in fact the query ran a little faster on the unpartitioned dataset. I guess that means that hive does not take advantage of partitions in this case.

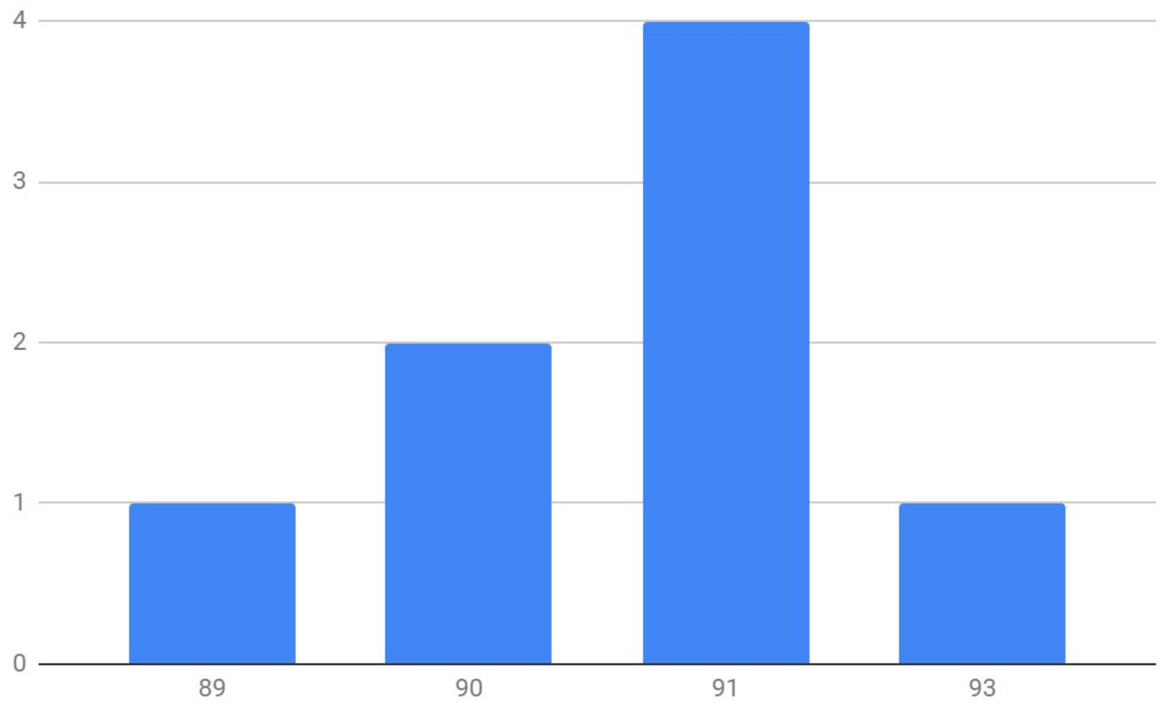
In addition to running this query on the 1.5 gig dataset, I also ran it on a larger 8 gig dataset, and the results were about 600 seconds for unpartitioned and about 640 seconds for partitioned.

Bucketing by cityid.

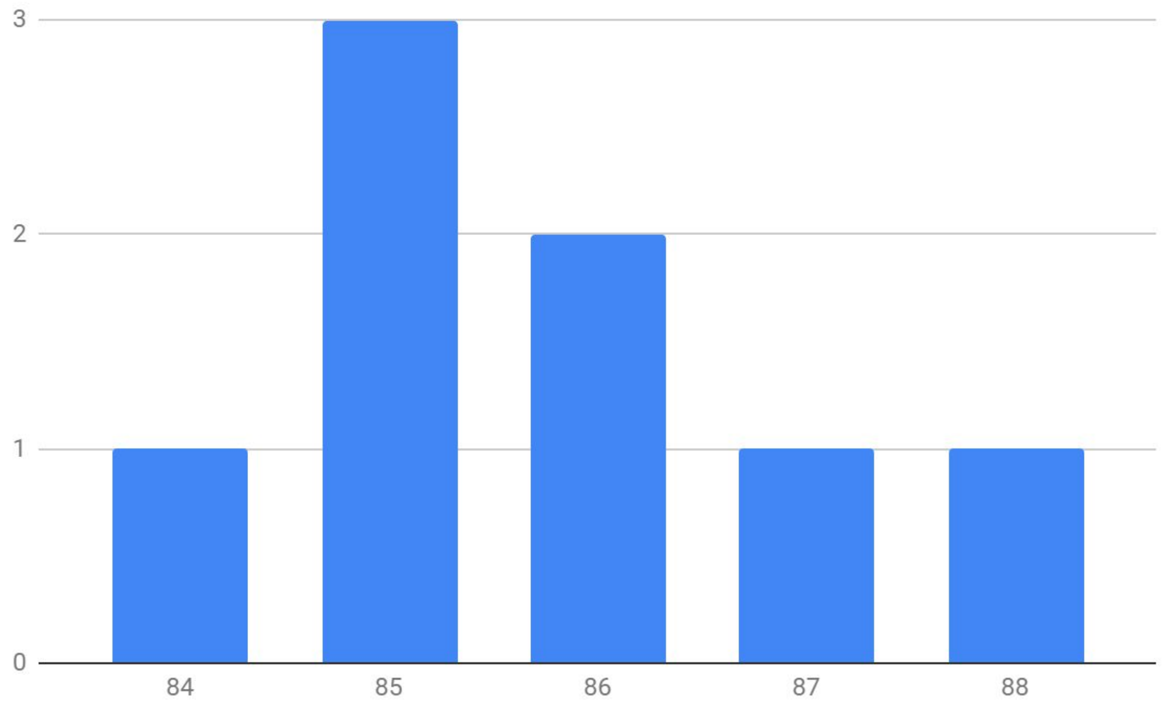
Since hive was not smart enough to take advantage of data being partitioned by **cityid**, I wonder if it can take advantage of our dataset being *bucketed* by **cityid**. Here are the results of my runs:

Attempt #	Unbucketed time (in seconds)	Bucketed time (in seconds)
1	91	86
2	90	85
3	91	86
4	90	84
5	91	85
6	91	85
7	89	88
8	93	87
Average	**90,75**	**85,75**

Unbucketed distribution:



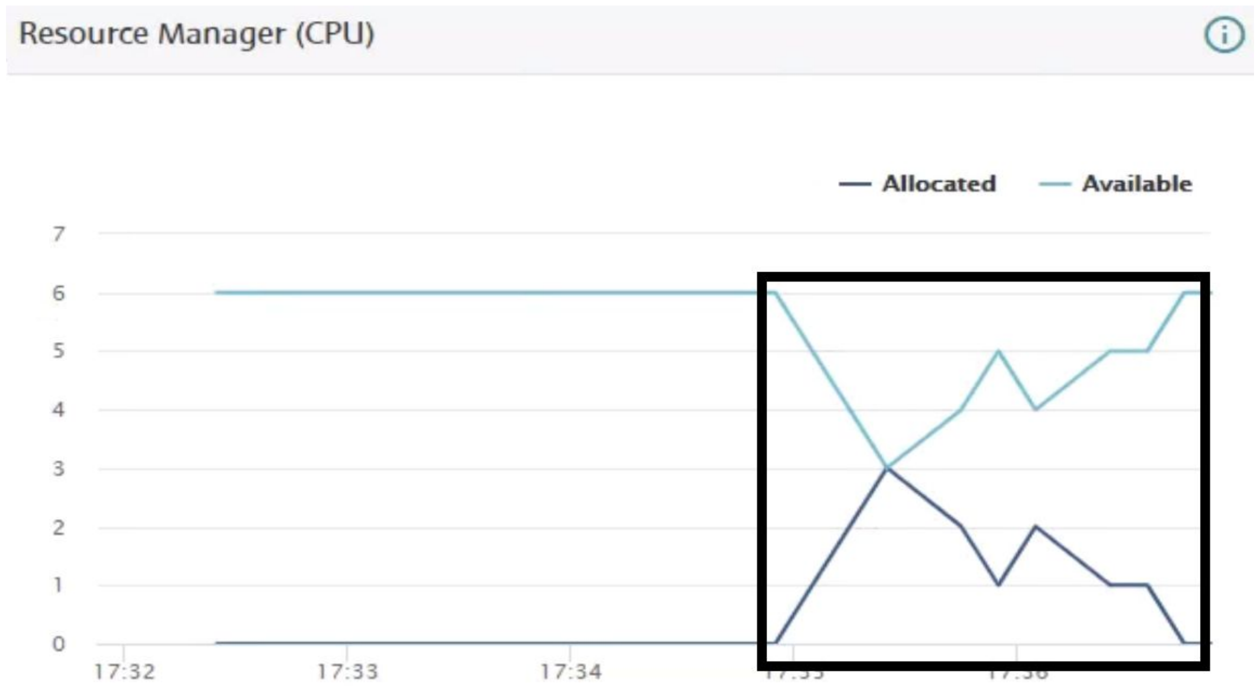
Bucketed distribution:

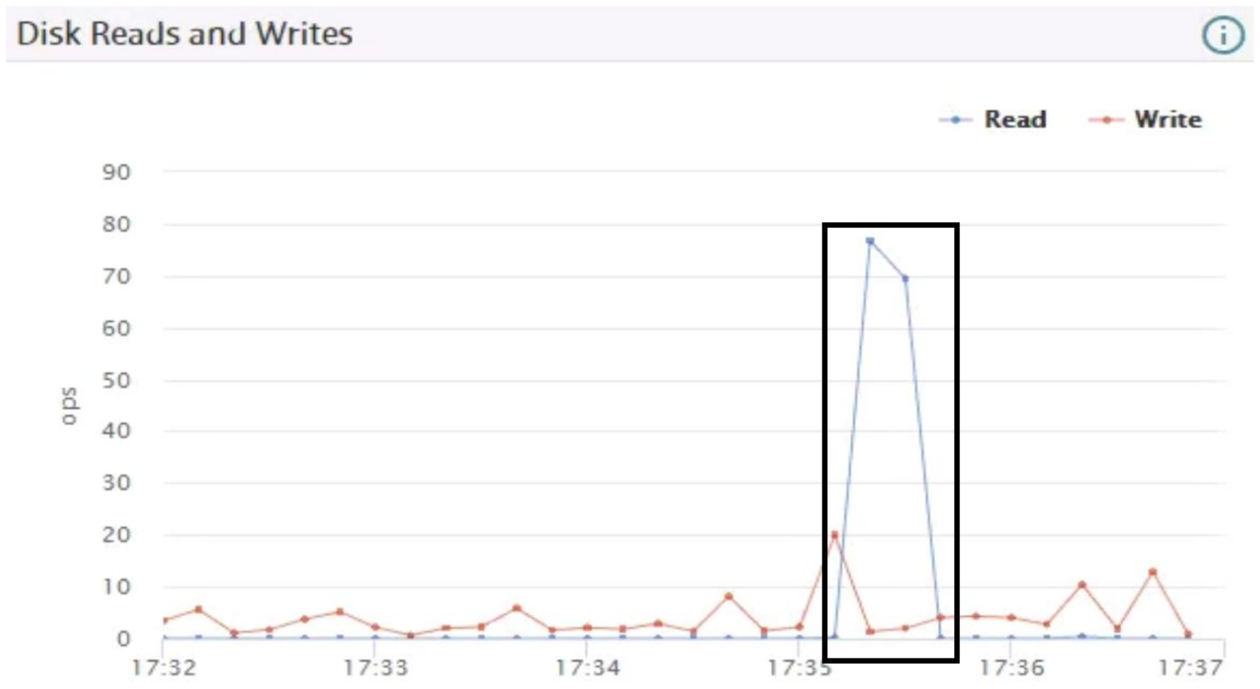


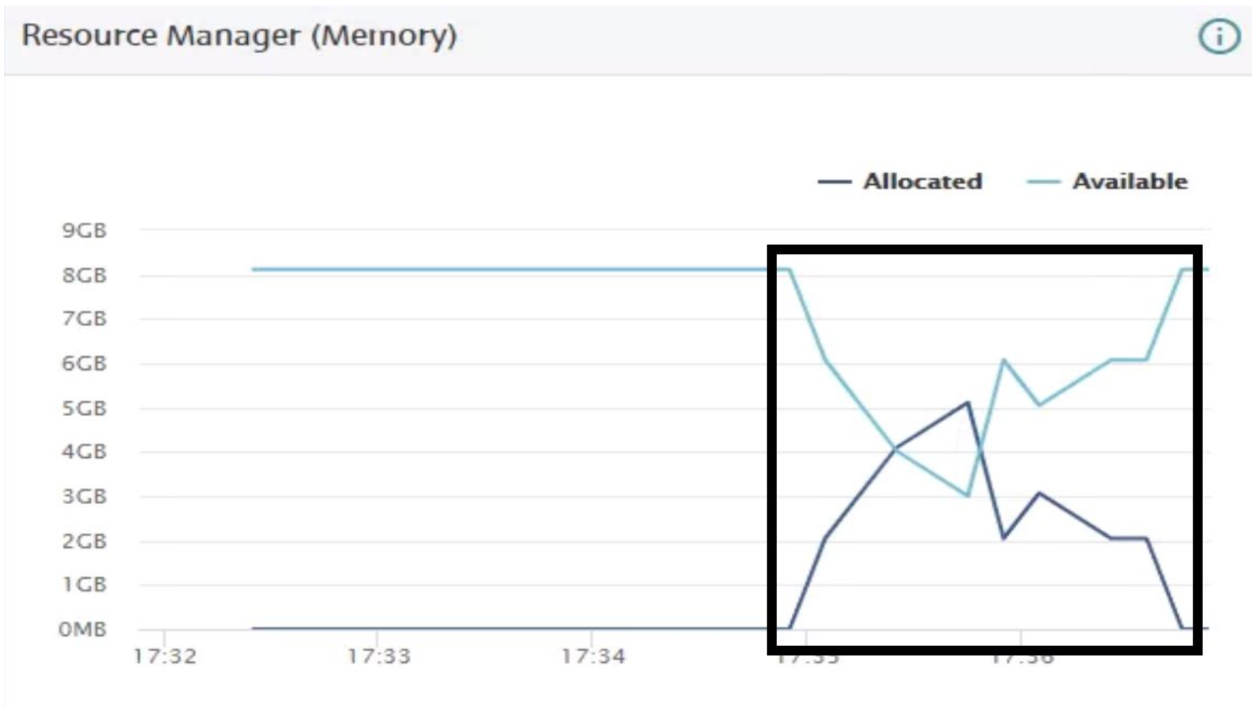
In addition to this ***it took about 175 seconds to bucket this dataset****.*

It’s easy to see that my query performs a little faster on bucketed data. Note that I bucketed my data into 60 buckets, and each bucket is approximately 25 mb. I made sure to take advantage of mapside-joins by setting hive.mapjoin.smalltable.filesize to a higher value that 25 mb as well.

Metrics:







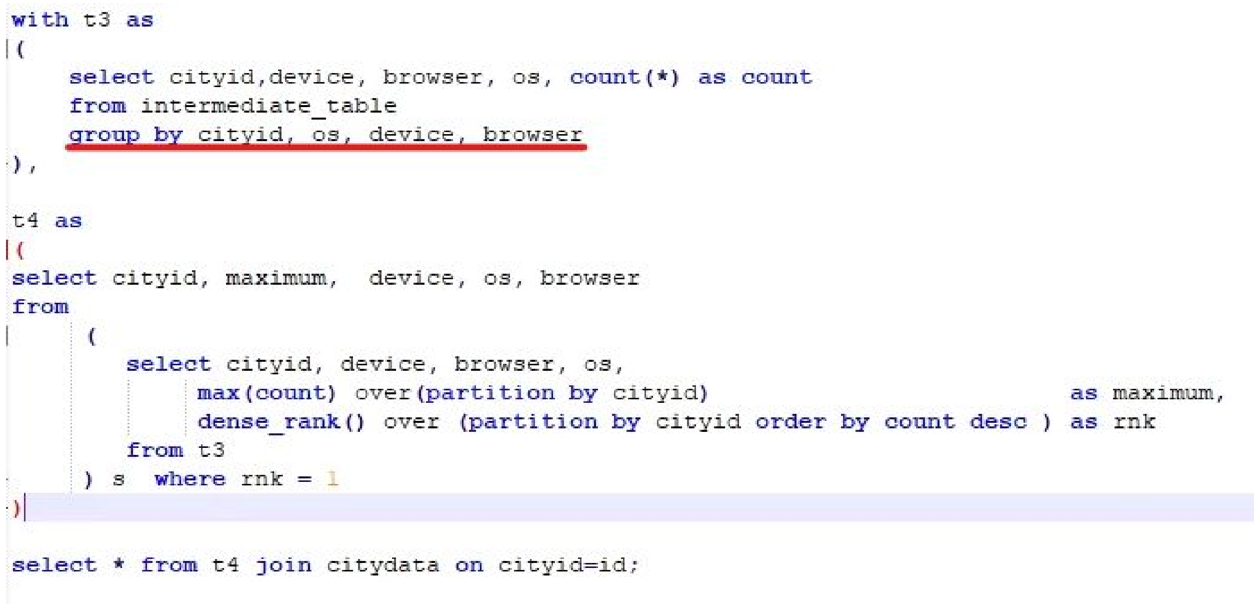
Once again the disk usage was much more dramatic than all the other factors.

Conclusion.

It seems like hive did take advantage of our dataset being bucketed by **cityid**. The performance difference is not dramatic, but it’s still noticeable.

Bucketing by cityid, device, os and browser.

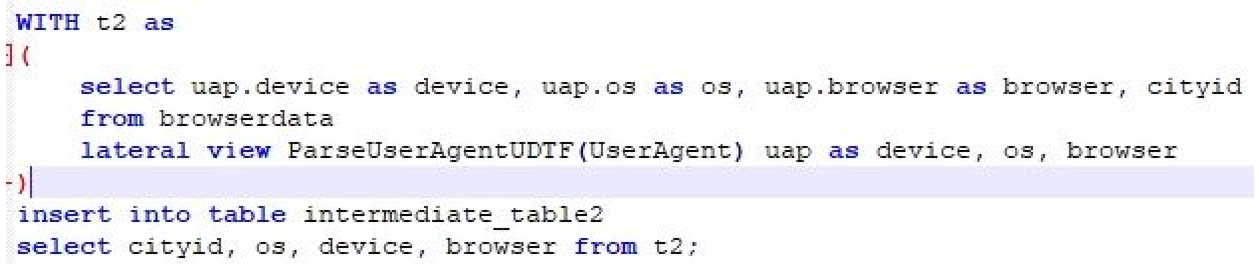
Since our original table does not include **device**, **browser** and **os** columns, we have to introduce an intermediate table in order to be able to use them. My query has an expensive group by operation that groups data into **<cityid, device, os, browser>** bins.



```
with t3 as
(
    select cityid,device, browser, os, count(*) as count
    from intermediate_table
    group by cityid, os, device, browser
),

t4 as
(
select cityid, maximum,  device, os, browser
from
    (
        select cityid, device, browser, os,
            max(count) over(partition by cityid)                            as maximum,
            dense_rank() over (partition by cityid order by count desc )  as rnk
        from t3
    ) s  where rnk = 1
)

select * from t4 join citydata on cityid=id;
```

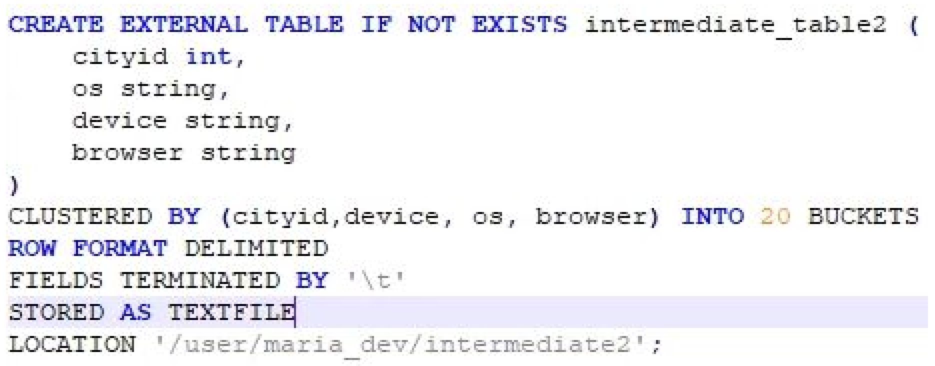
Note that we have 1 fewer operations in this query, because we already executed the t2 step from the original table, while creating the intermediate table.

Here’s the t2 step that I am talking about (we use it to write into the intermediate table):



```
WITH t2 as
(
    select uap.device as device, uap.os as os, uap.browser as browser, cityid
    from browserdata
    lateral view ParseUserAgentUDTF(UserAgent) uap as device, os, browser
)
insert into table intermediate_table2
select cityid, os, device, browser from t2;
```

Here’s the definition of this table:



```
CREATE EXTERNAL TABLE IF NOT EXISTS intermediate_table2 (
    cityid int,
    os string,
    device string,
    browser string
)
CLUSTERED BY (cityid,device, os, browser) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/user/maria_dev/intermediate2';
```

I also created 2 similar tables, one with 50 buckets and one with just 6 buckets, to see if the number of buckets affects performance, and the results of 20 vs 6 vs 50 were pretty much the same, so I decided to go with 20 buckets in the long run (the size of my data is a bit less than 1.5 gigs).

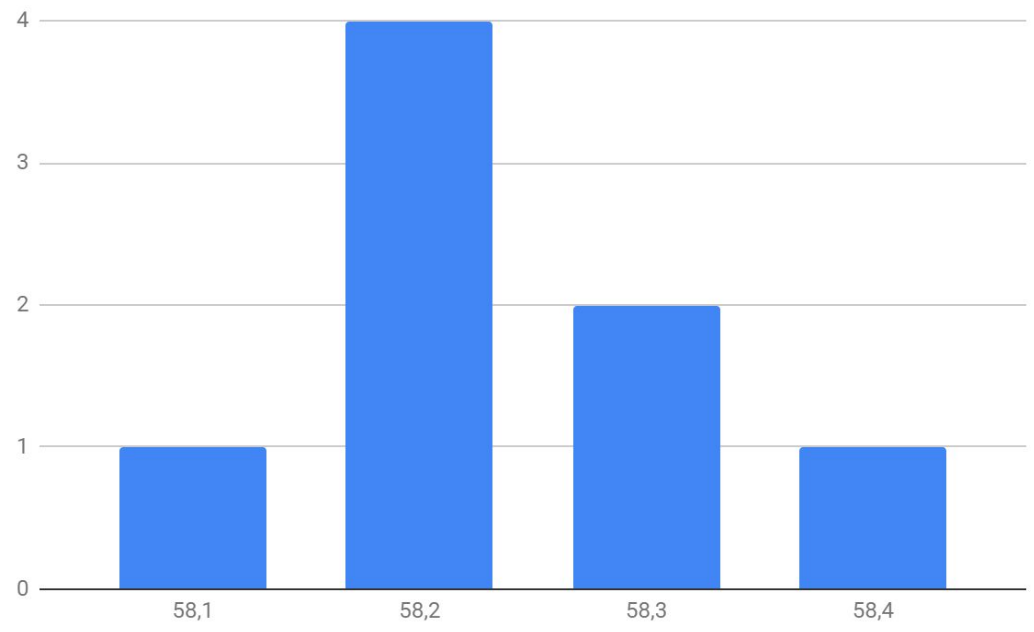
In general, as this link suggests, if the number of buckets in a schema does not affect performance that much, it is best to choose the number of buckets such that **dataset_size** / **block_size** = **bucket_size** so that our bucket size is roughly the same as the block size on our file system.

- In case of MapR-FS the block size is 256mb, so there should be 1.5 gig / 256 = 6 buckets.
- In case of cloudera's HDFS it's 128mb, so there should be 12 buckets.
- In case of vanilla hadoop it's 64 mb, so there should be 24 buckets.

Here are the results I got:

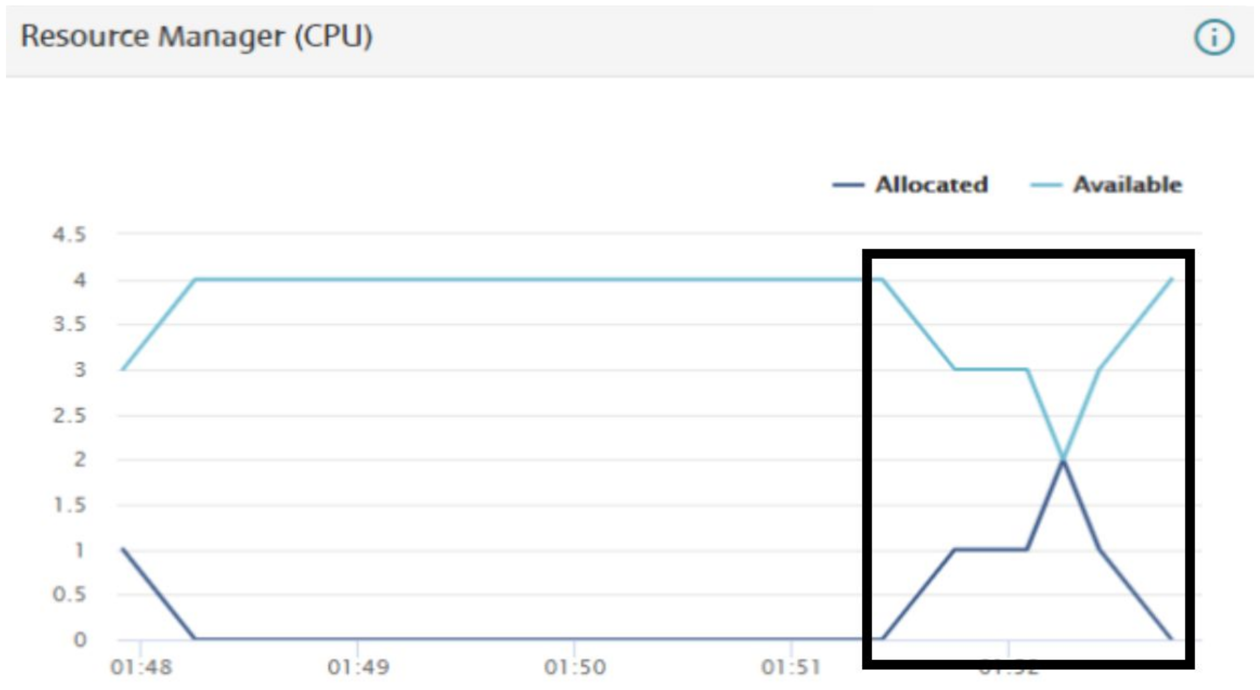
Attempt #	Run time (in seconds)
1	58.2
2	58.2
3	58.2
4	58.3
5	58.4
6	58.1
7	58.2
8	58.3
Average	**58,2375**

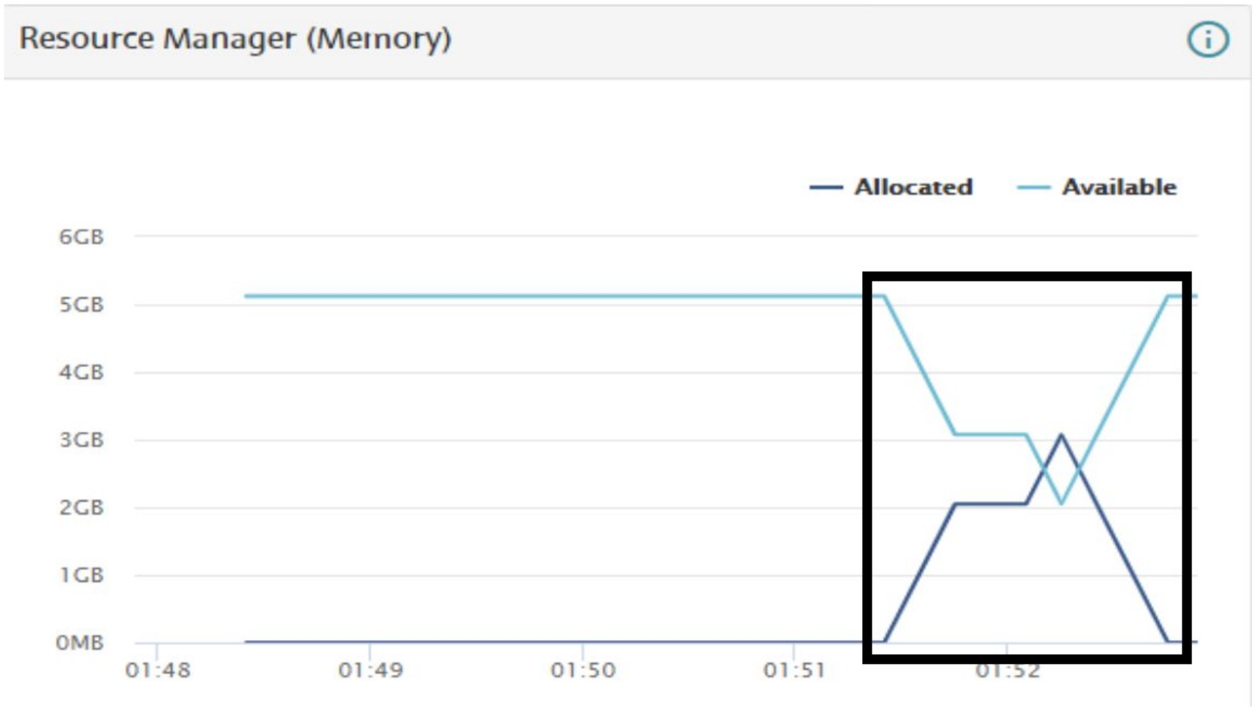
Distribution:

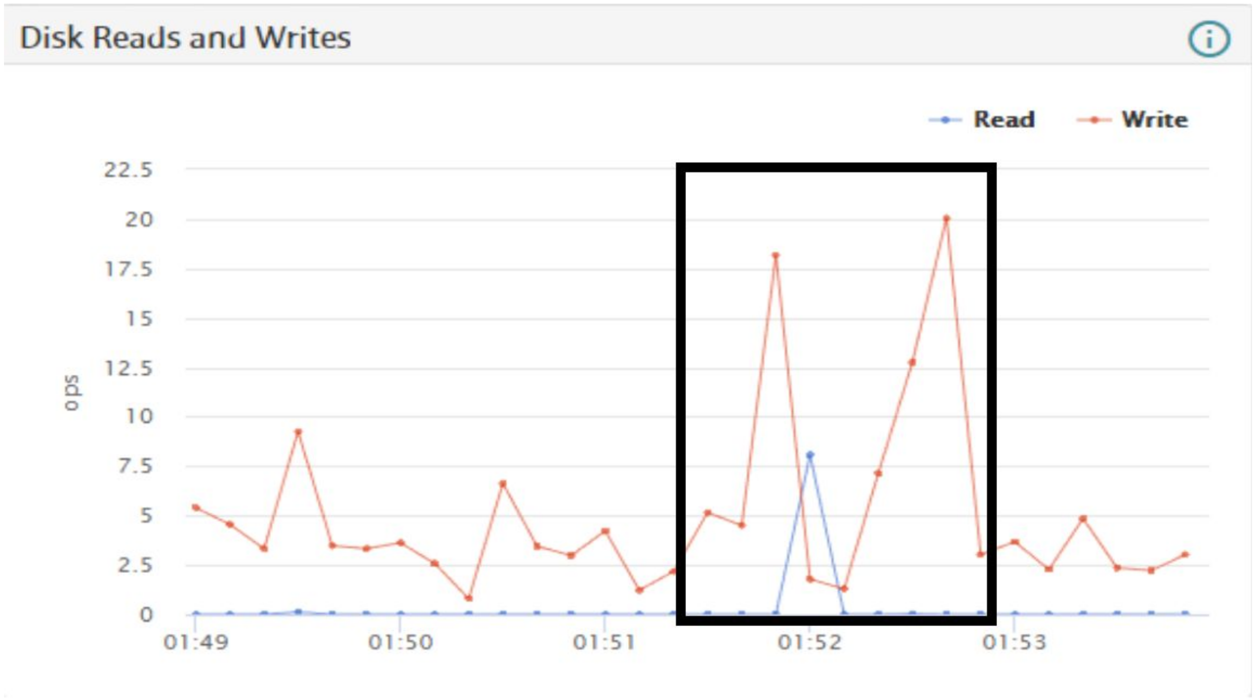


It took about 172 seconds to write into this bucketed table.

Metrics:







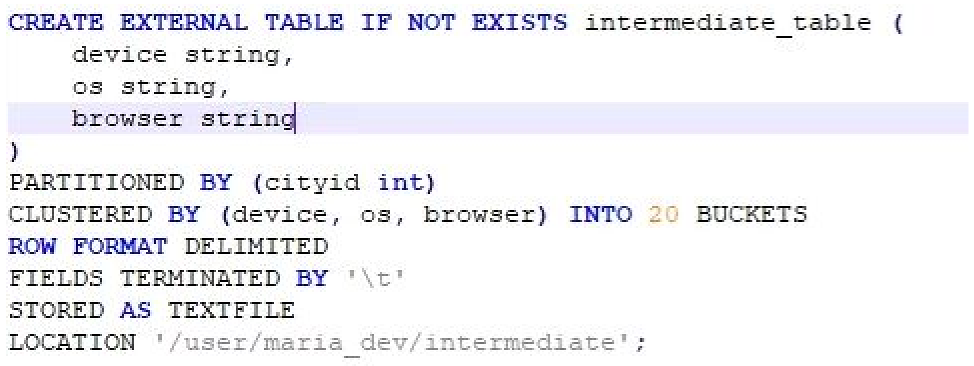
This one didn’t seem to have a limiting factor at all and ran pretty quick. So it seems much more scalable than the others.

Conclusion.

This is by far the best performing schema of all.

Partitioning by cityid and bucketing by device, os and browser.

Now let's see how it's going to fare if we partition our data by **cityid** and bucket it by **device**, **os** and **browser**. An intermediate table will be used once again. Here's the new table's definition:

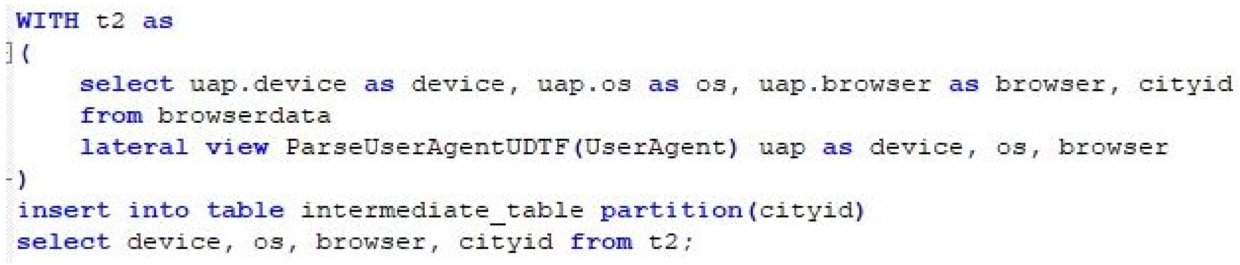


```
CREATE EXTERNAL TABLE IF NOT EXISTS intermediate_table (
    device string,
    os string,
    browser string
)
PARTITIONED BY (cityid int)
CLUSTERED BY (device, os, browser) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/user/maria_dev/intermediate';
```

I decided to distribute data into 20 buckets like in the last query, because the number of buckets didn't seem to affect performance at all.

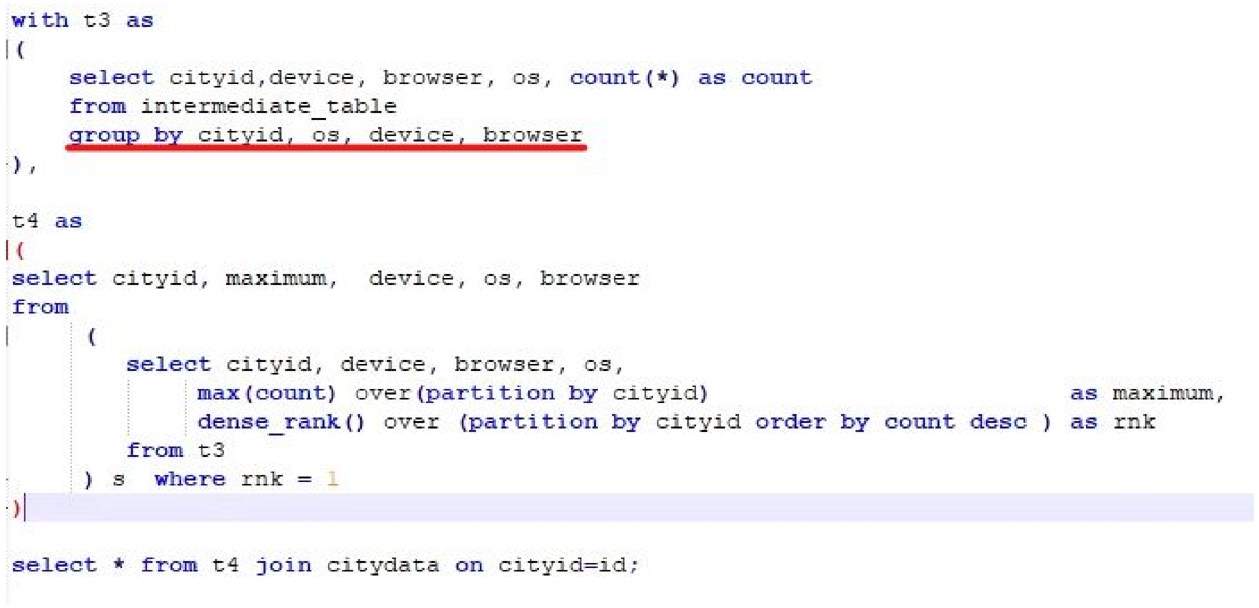
As well as in the previous schema, I am hoping to optimize that group by in t3 as well as the operations that work with cityid.

Here's the query I use to insert into the new table:



```
WITH t2 as
(
    select uap.device as device, uap.os as os, uap.browser as browser, cityid
    from browserdata
    lateral view ParseUserAgentUDTF(UserAgent) uap as device, os, browser
)
insert into table intermediate_table partition(cityid)
select device, os, browser, cityid from t2;
```

And here's the query that I run to calculate the final result:



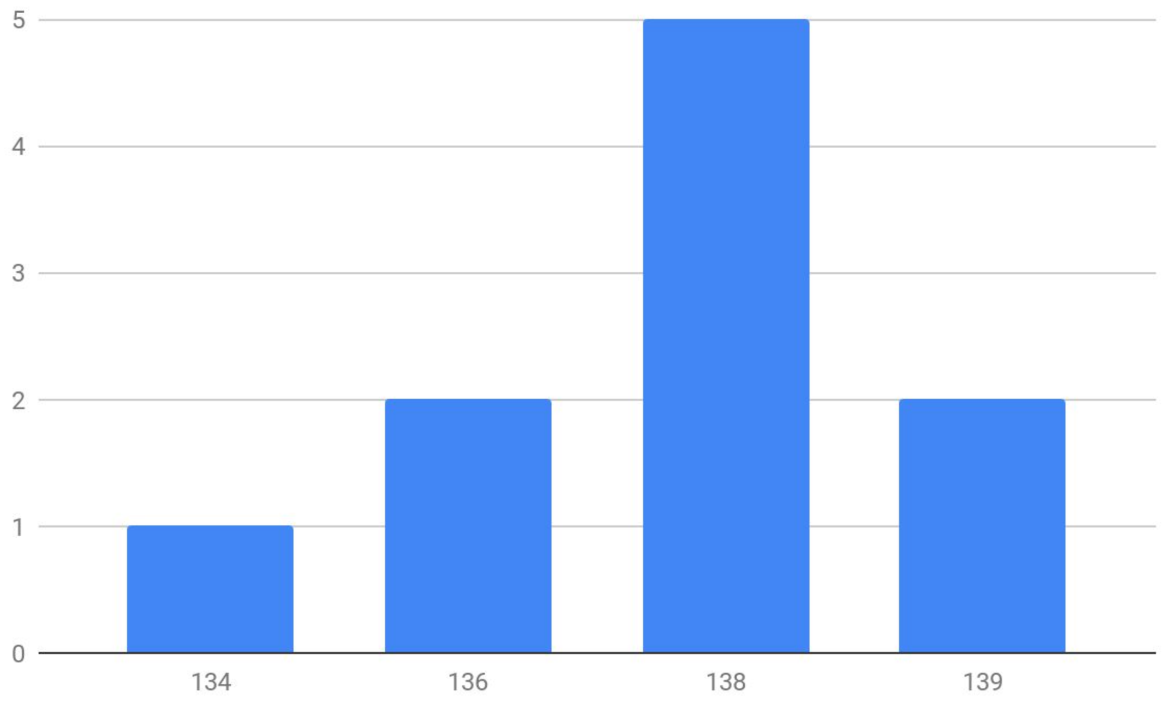
```
with t3 as
(
    select cityid,device, browser, os, count(*) as count
    from intermediate_table
    group by cityid, os, device, browser
),

t4 as
(
select cityid, maximum,  device, os, browser
from
      (
        select cityid, device, browser, os,
              max(count) over(partition by cityid)                         as maximum,
              dense_rank() over (partition by cityid order by count desc ) as rnk
        from t3
      ) s  where rnk = 1
)

select * from t4 join citydata on cityid=id;
```

Note that we have 1 fewer operations, because we already executed the t2 step from the original table, while creating the intermediate table.

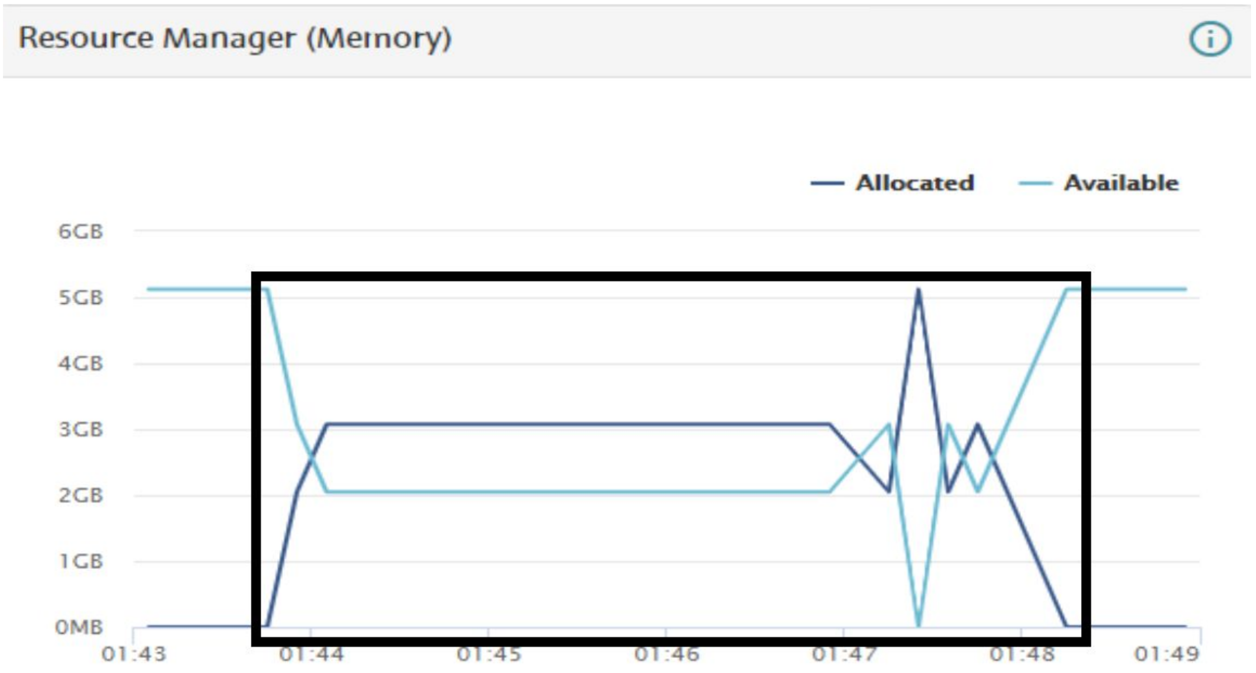
Let’s look at the execution results:

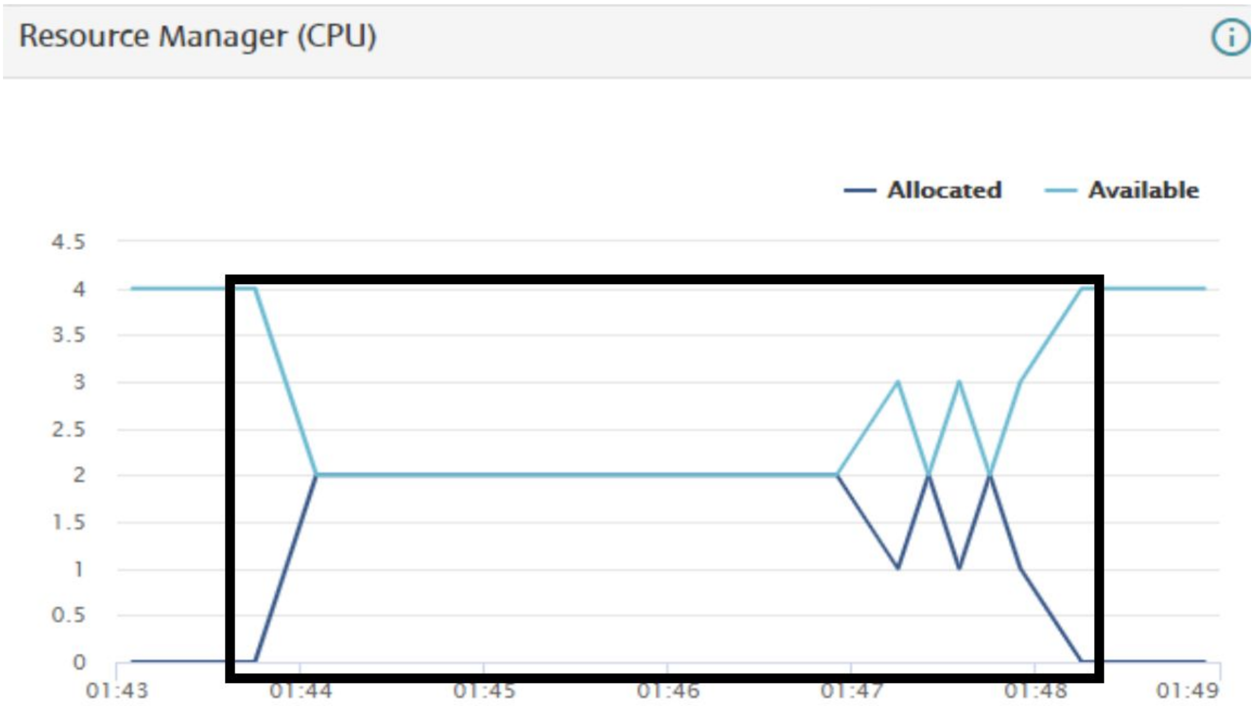
Attempt #	Run time against the intermediate table (in seconds)
1	134
2	137
3	136
4	136
5	138
6	138
7	138
8	139
9	138
10	138
Average	**137,2**

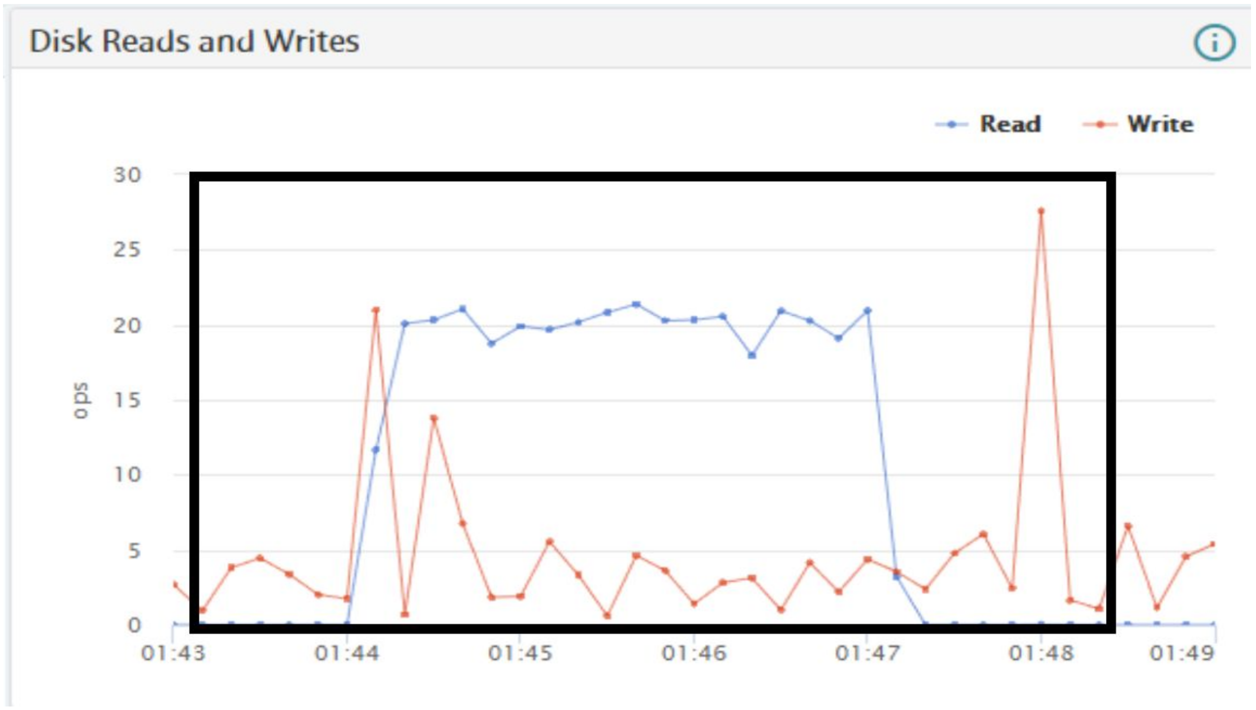
Distribution:



It took about 184 seconds on average to write into this table.


Resource Manager (Memory)
Allocated
Available
6GB
5GB
4GB
3GB
2GB
1GB
0MB
01:43
01:44
01:45
01:46
01:47
01:48
01:49


Resource Manager (CPU)
Allocated
Available
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
01:43
01:44
01:45
01:46
01:47
01:48
01:49



It seems that RAM was the main limiting factor here.

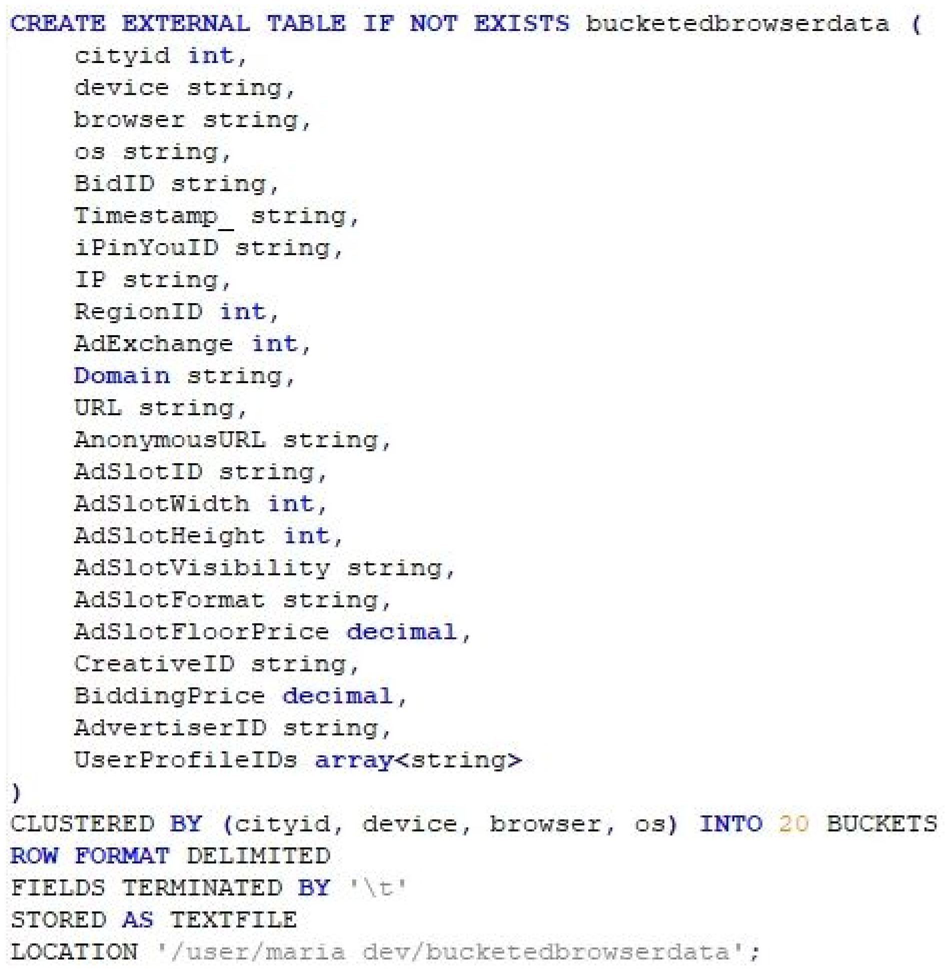
Conclusion.

The result is absolutely mind-boggling, because one would think that having all these buckets and partitions is supposed to help, but it only slows down the performance.

Bucketing the whole table and not just individual columns.

I declared the schema from ***Bucketing by cityid, device, os and browser*** to be our winner. But one might argue that the comparison was unfair, because I was only using an excerpt from the original table, and I also used an intermediate table. But what if our whole table what initially in a bit different format? Let's now try to put the whole table to work.

Here's what the schema looks like:



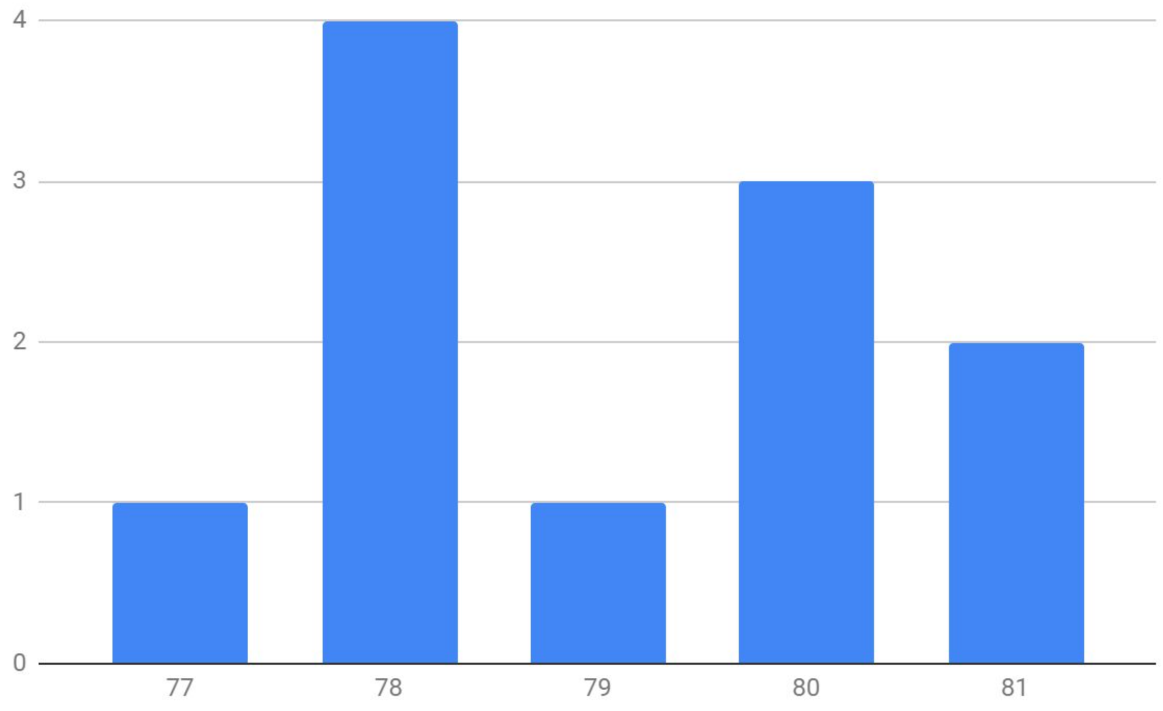
```
CREATE EXTERNAL TABLE IF NOT EXISTS bucketedbrowserdata (
    cityid int,
    device string,
    browser string,
    os string,
    BidID string,
    Timestamp_ string,
    iPinYouID string,
    IP string,
    RegionID int,
    AdExchange int,
    Domain string,
    URL string,
    AnonymousURL string,
    AdSlotID string,
    AdSlotWidth int,
    AdSlotHeight int,
    AdSlotVisibility string,
    AdSlotFormat string,
    AdSlotFloorPrice decimal,
    CreativeID string,
    BiddingPrice decimal,
    AdvertiserID string,
    UserProfileIDs array<string>
)
CLUSTERED BY (cityid, device, browser, os) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/user/maria dev/bucketedbrowserdata';
```

Here are the results I got with this schema:

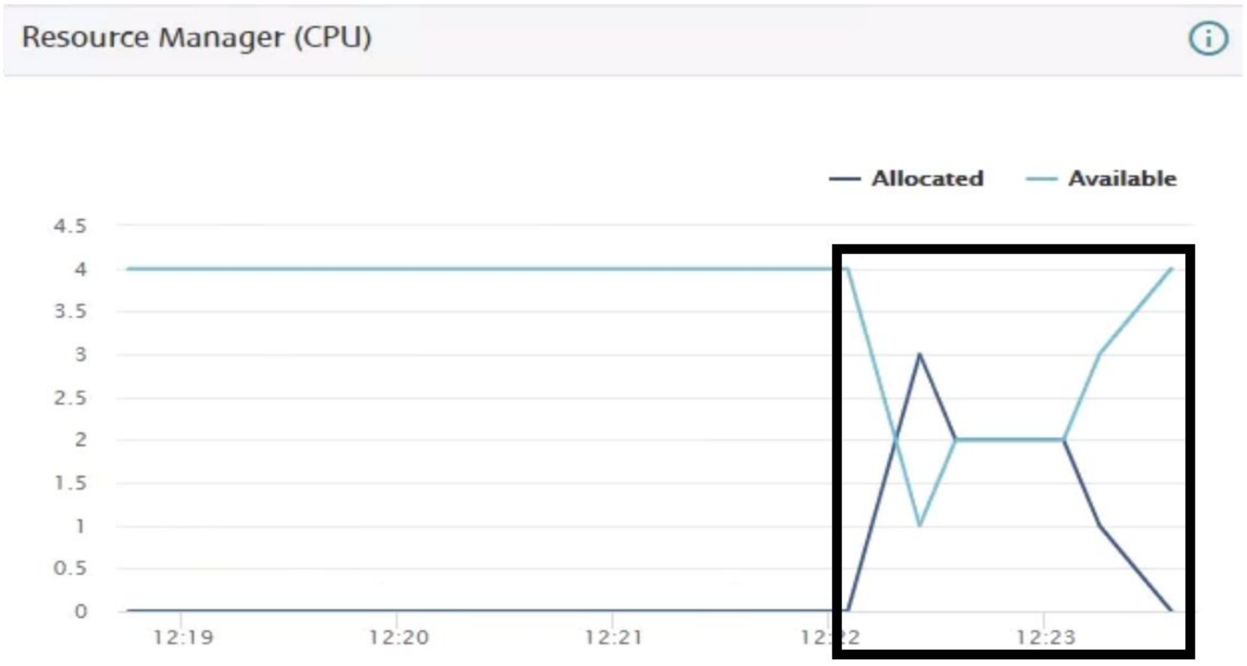
Attempt #	Run time (in seconds)
1	81
2	79
3	80

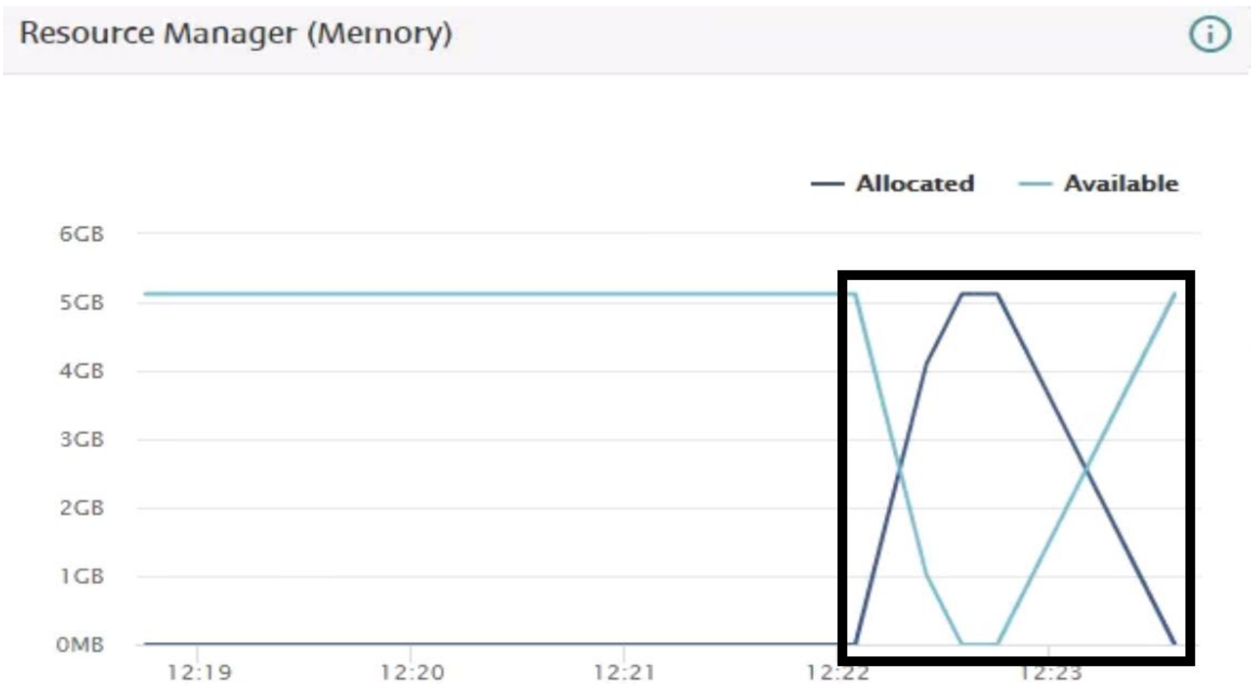
4	78
5	81
6	80
7	77
8	78
9	78
10	80
11	78
Average	**79,(09)**

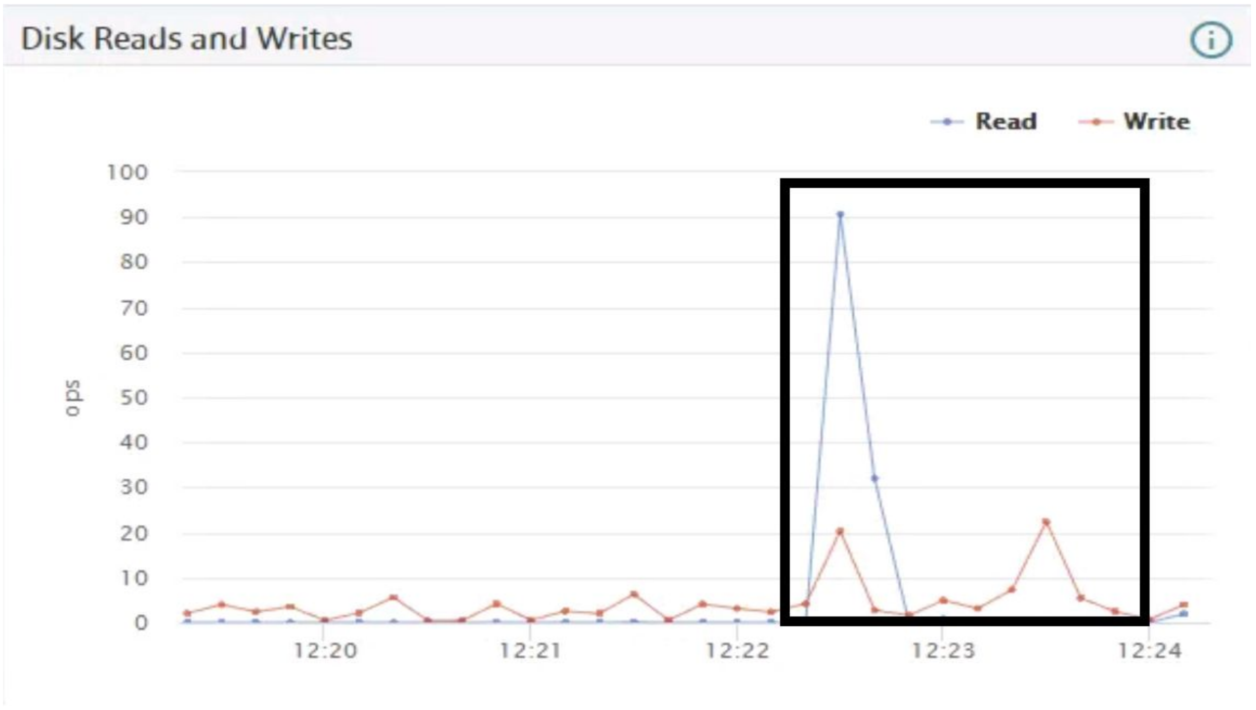
Distribution:



It took about 177 seconds to write into this table.


Resource Manager (CPU)
Allocated
Available
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
12:19
12:20
12:21
12:22
12:23


Resource Manager (Memory)
Allocated
Available
6GB
5GB
4GB
3GB
2GB
1GB
0MB
12:19
12:20
12:21
12:22
12:23



It’s easy to see that RAM was the main limiting factor.

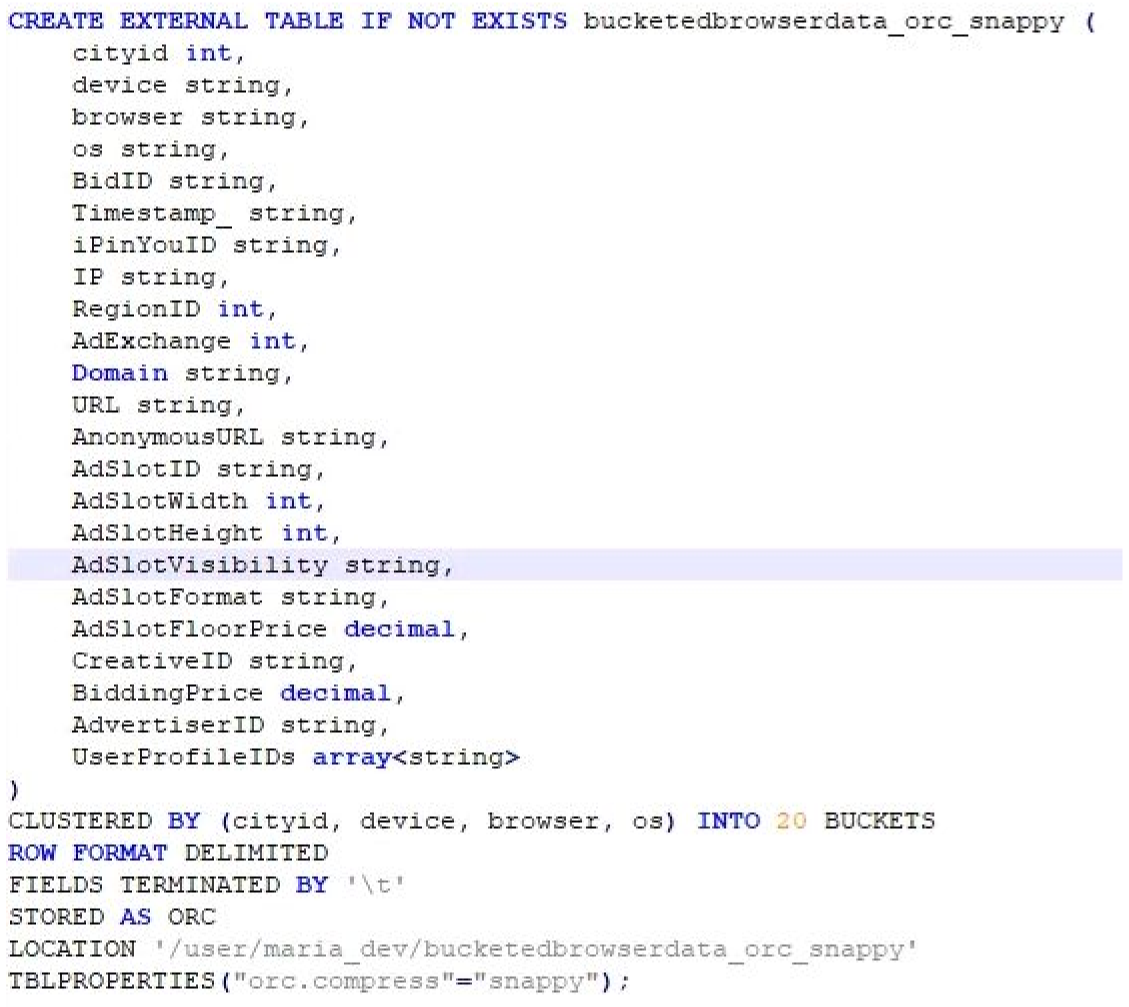
Conclusion.

Indeed it runs a bit slower than on the dataset that only contains the necessary columns, but it’s still faster than all the other schemas, this is why I am gonna choose this schema as the best one and continue working with it.

Compression.

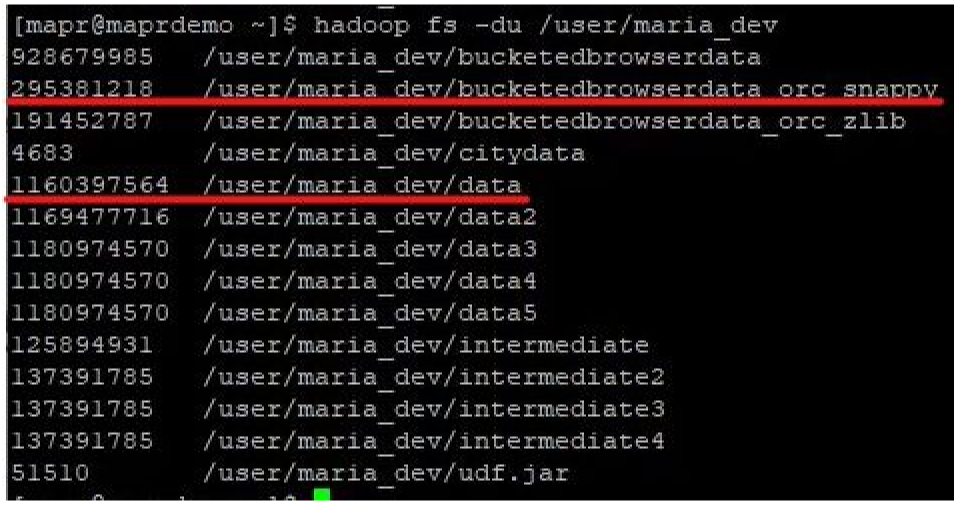
ORC format with snappy compression.

Here is the schema that I will use to store our data in the ORC format with the “snappy” compression.



```
CREATE EXTERNAL TABLE IF NOT EXISTS bucketedbrowserdata_orc_snappy (
    cityid int,
    device string,
    browser string,
    os string,
    BidID string,
    Timestamp_ string,
    iPinYouID string,
    IP string,
    RegionID int,
    AdExchange int,
    Domain string,
    URL string,
    AnonymousURL string,
    AdSlotID string,
    AdSlotWidth int,
    AdSlotHeight int,
    AdSlotVisibility string,
    AdSlotFormat string,
    AdSlotFloorPrice decimal,
    CreativeID string,
    BiddingPrice decimal,
    AdvertiserID string,
    UserProfileIDs array<string>
)
CLUSTERED BY (cityid, device, browser, os) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS ORC
LOCATION '/user/maria_dev/bucketedbrowserdata_orc_snappy'
TBLPROPERTIES("orc.compress"="snappy");
```

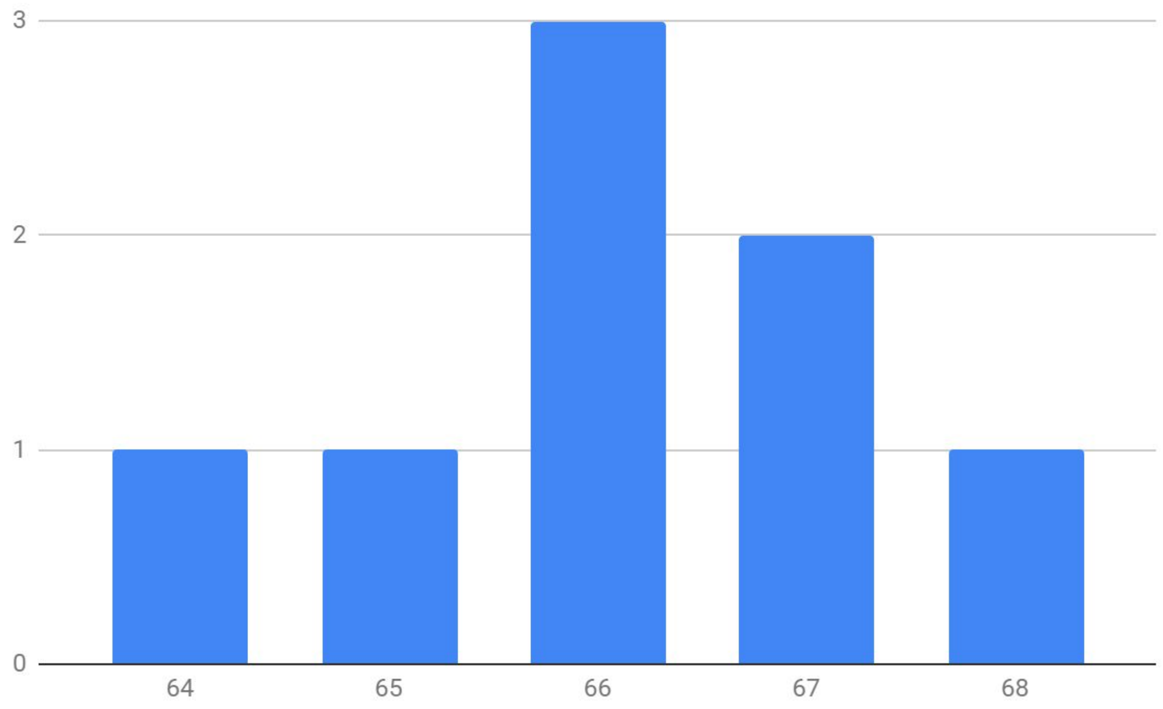
Here’s a screenshot showing the sizes of the original data and compressed data:



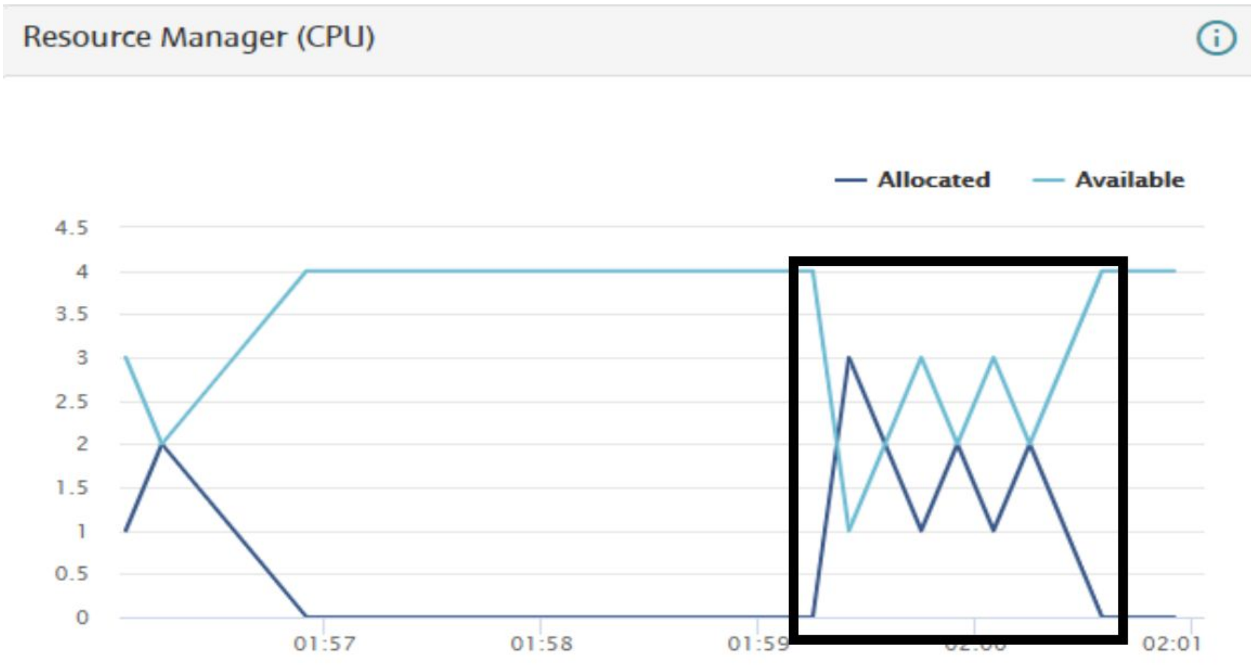
```
[mapr@maprdemo ~]$ hadoop fs -du /user/maria_dev
928679985   /user/maria_dev/bucketedbrowserdata
295381218   /user/maria_dev/bucketedbrowserdata_orc_snappy
191452787   /user/maria_dev/bucketedbrowserdata_orc_zlib
4683        /user/maria_dev/citydata
1160397564  /user/maria_dev/data
1169477716  /user/maria_dev/data2
1180974570  /user/maria_dev/data3
1180974570  /user/maria_dev/data4
1180974570  /user/maria_dev/data5
125894931   /user/maria_dev/intermediate
137391785   /user/maria_dev/intermediate2
137391785   /user/maria_dev/intermediate3
137391785   /user/maria_dev/intermediate4
51510       /user/maria_dev/udf.jar
```

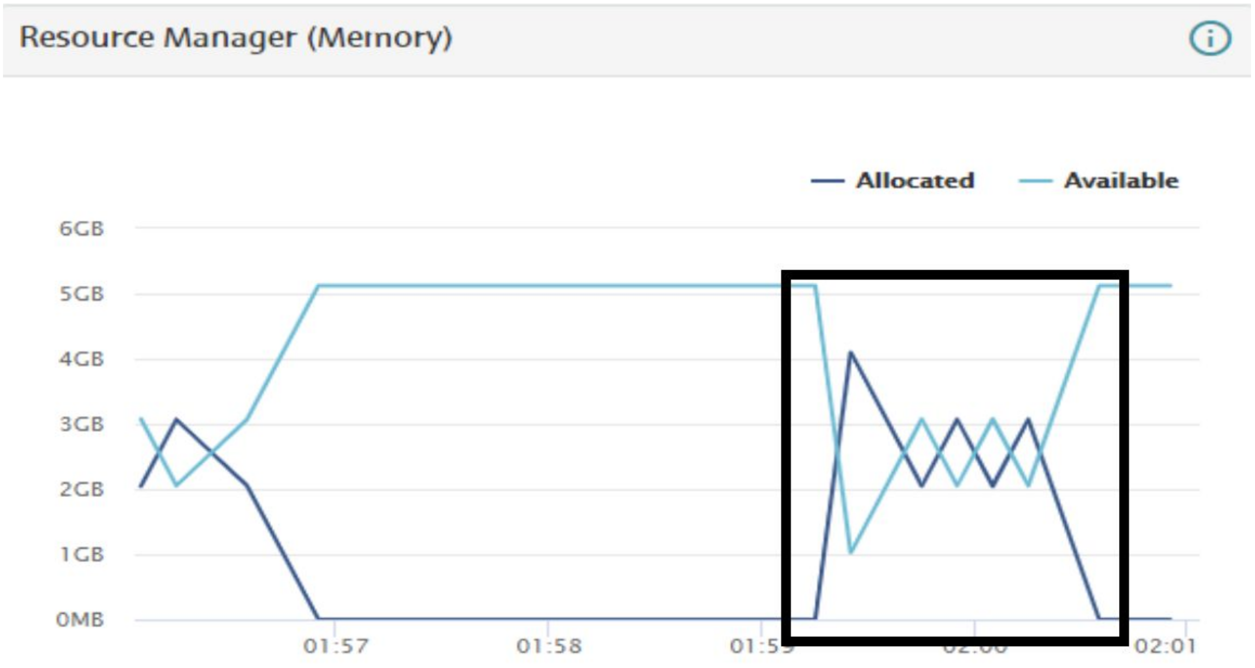
As you can see we went from 1 gigabyte (with change) to 295 megabytes.
Query run results:

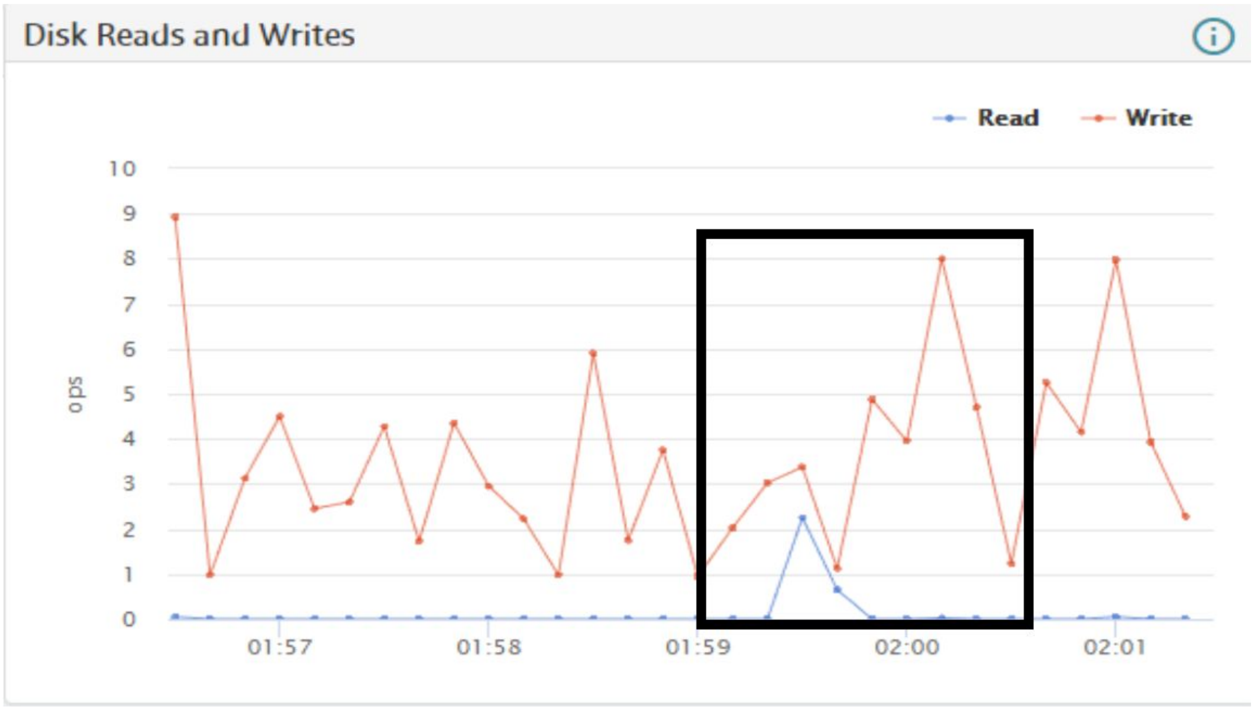
Attempt #	Run time (in seconds)
1	67
2	66
3	64
4	68
5	66
6	66
7	65
8	67
Average	**66,125**



It took about 208 seconds to write into this table.

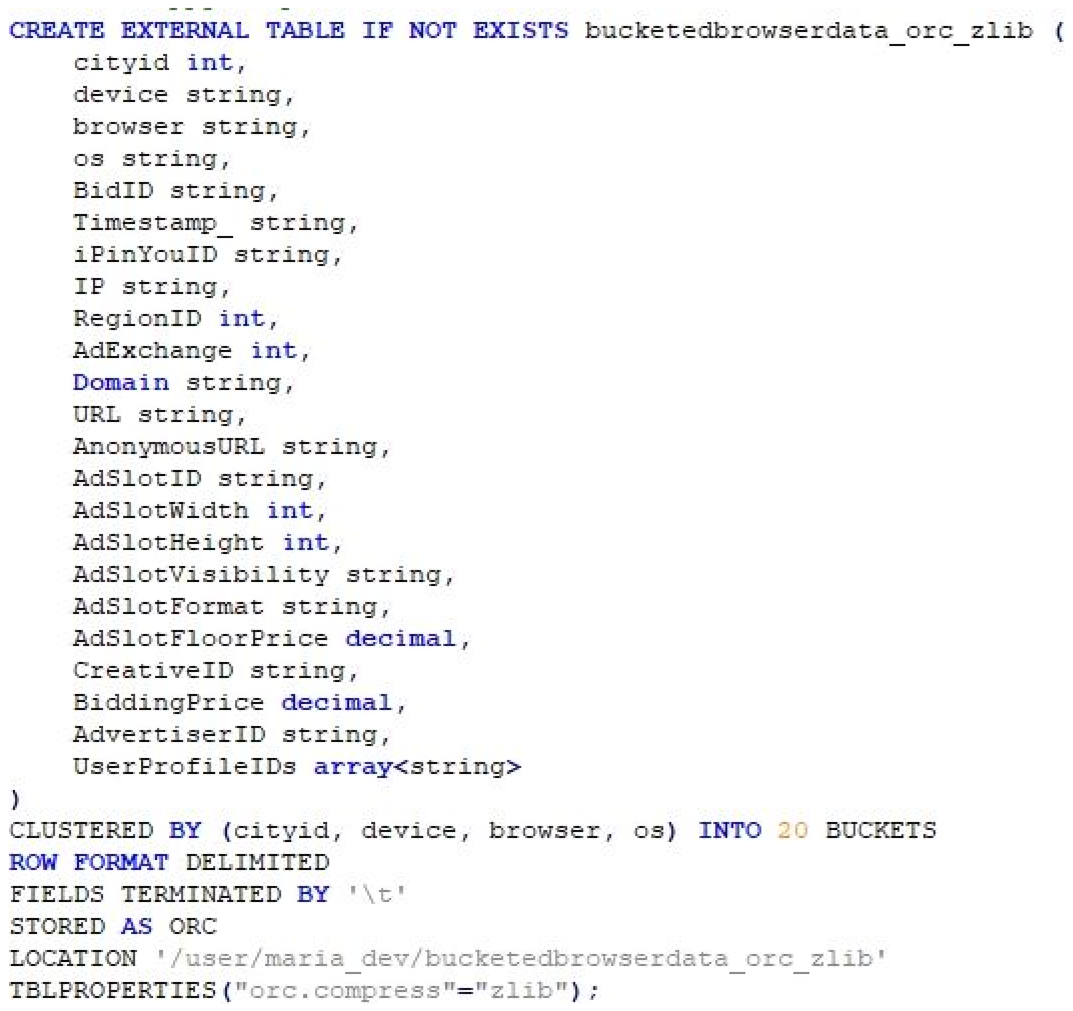

Resource Manager (CPU)
Allocated
Available
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
01:57
01:58
01:59
02:00
02:01


Resource Manager (Memory)
Allocated
Available
6GB
5GB
4GB
3GB
2GB
1GB
0MB
01:57
01:58
01:59
02:00
02:01


Disk Reads and Writes
Read
Write
ops
10
9
8
7
6
5
4
3
2
1
0
01:57
01:58
01:59
02:00
02:01

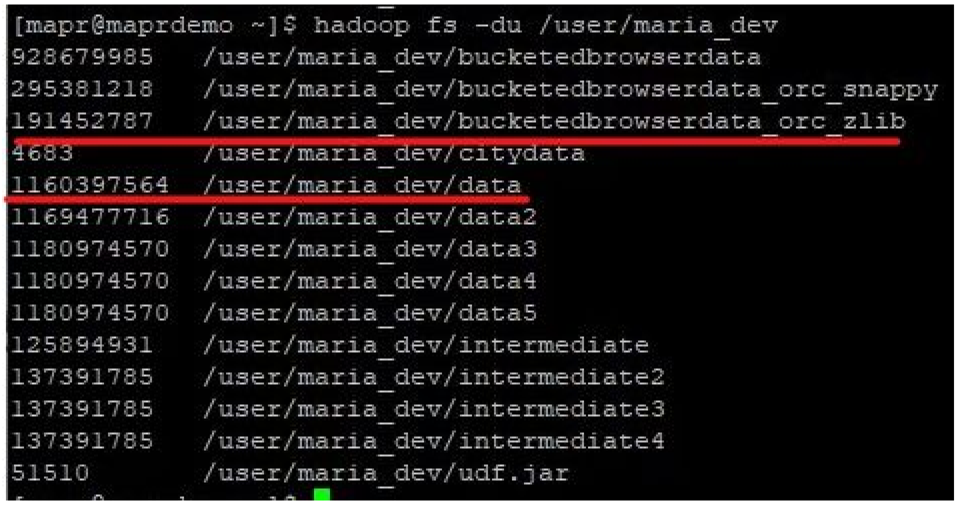
ORC format with zlib compression.

Here is the schema that I will use to store our data in the ORC format with the “zlib” compression:



```
CREATE EXTERNAL TABLE IF NOT EXISTS bucketedbrowserdata_orc_zlib (
    cityid int,
    device string,
    browser string,
    os string,
    BidID string,
    Timestamp_ string,
    iPinYouID string,
    IP string,
    RegionID int,
    AdExchange int,
    Domain string,
    URL string,
    AnonymousURL string,
    AdSlotID string,
    AdSlotWidth int,
    AdSlotHeight int,
    AdSlotVisibility string,
    AdSlotFormat string,
    AdSlotFloorPrice decimal,
    CreativeID string,
    BiddingPrice decimal,
    AdvertiserID string,
    UserProfileIDs array<string>
)
CLUSTERED BY (cityid, device, browser, os) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS ORC
LOCATION '/user/maria_dev/bucketedbrowserdata_orc_zlib'
TBLPROPERTIES("orc.compress"="zlib");
```

And here’s a screenshot comparing the sizes of the original data and the data compressed with zlib compression:

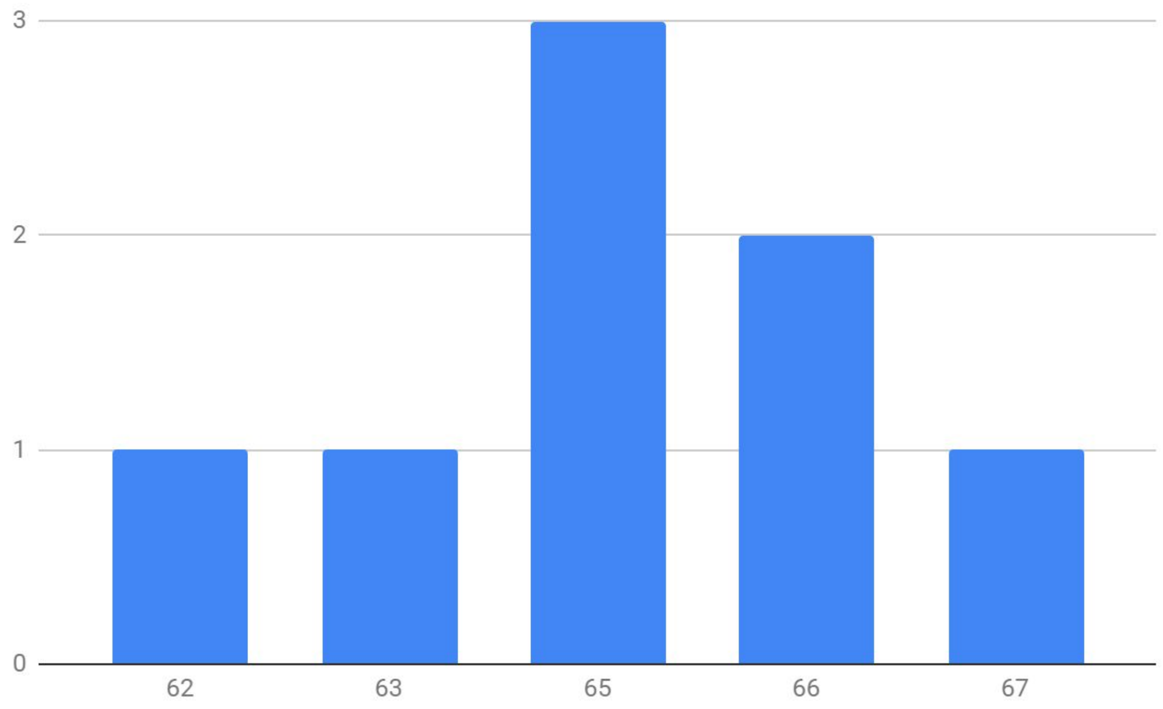


```
[mapr@maprdemo ~]$ hadoop fs -du /user/maria_dev
928679985   /user/maria_dev/bucketedbrowserdata
295381218   /user/maria_dev/bucketedbrowserdata_orc_snappy
191452787   /user/maria_dev/bucketedbrowserdata_orc_zlib
4683        /user/maria_dev/citydata
1160397564  /user/maria_dev/data
1169477716  /user/maria_dev/data2
1180974570  /user/maria_dev/data3
1180974570  /user/maria_dev/data4
1180974570  /user/maria_dev/data5
125894931   /user/maria_dev/intermediate
137391785   /user/maria_dev/intermediate2
137391785   /user/maria_dev/intermediate3
137391785   /user/maria_dev/intermediate4
51510       /user/maria_dev/udf.jar
```

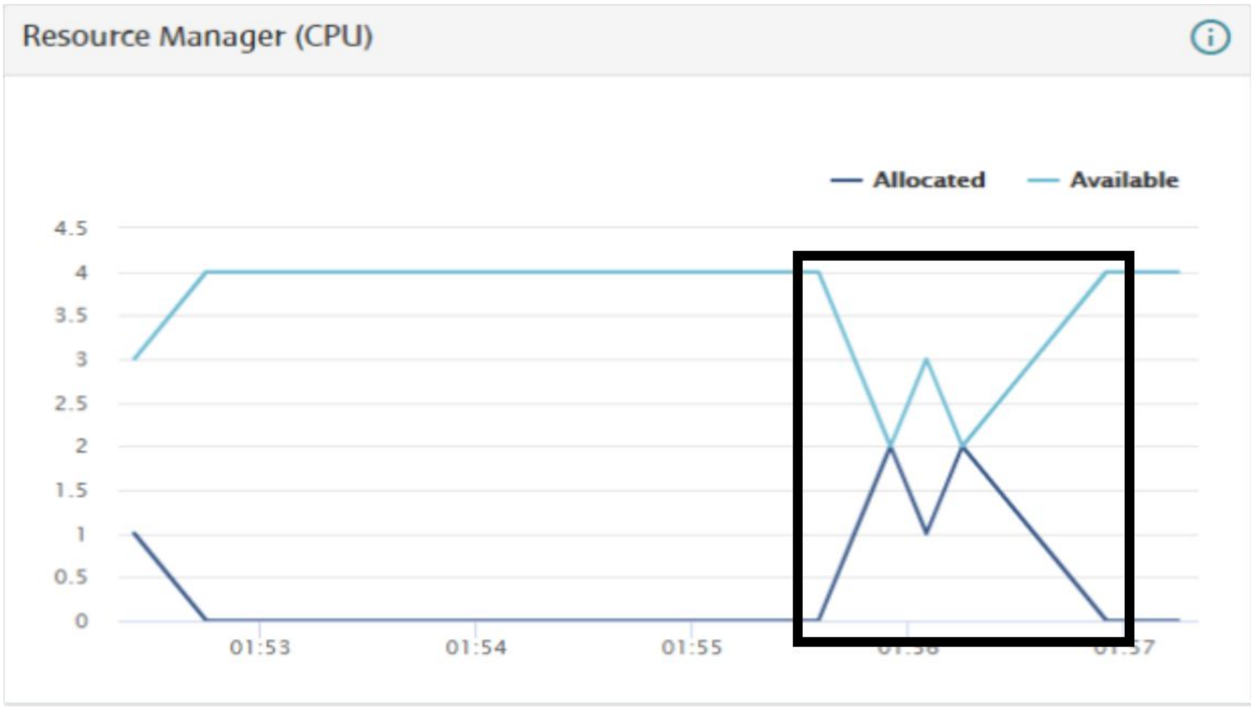
As you can see, we went from 1 gig to 191 megabytes. This is even better than snappy.

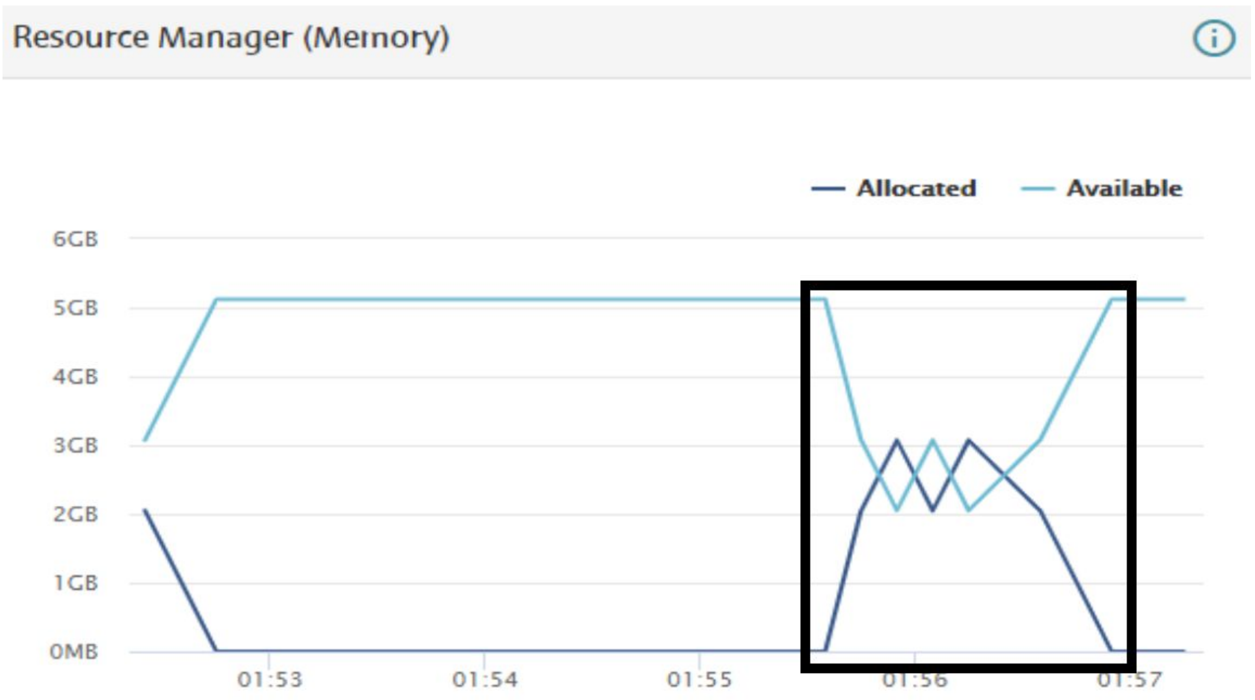
Query run results:

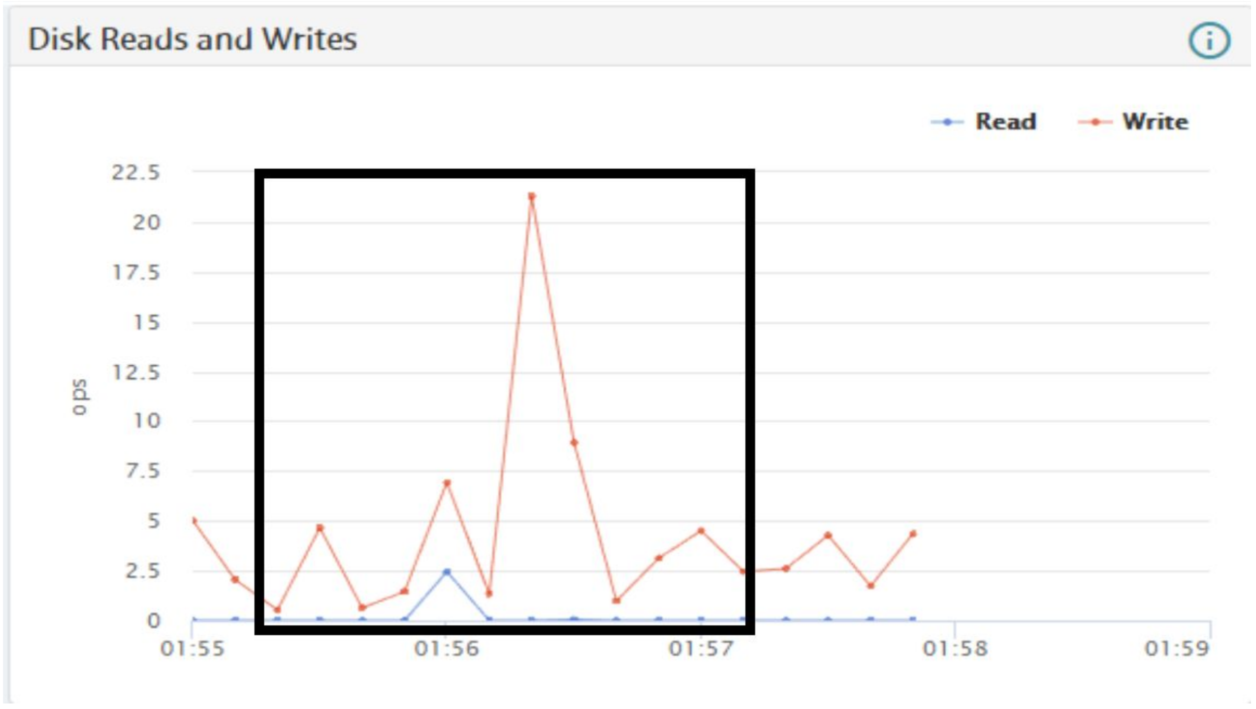
Attempt #	Run time (in seconds)
1	66
2	65
3	63
4	62
5	65
6	66
7	67
8	65
Average	**64,875**



***It took about 215 seconds to write into this table*.**

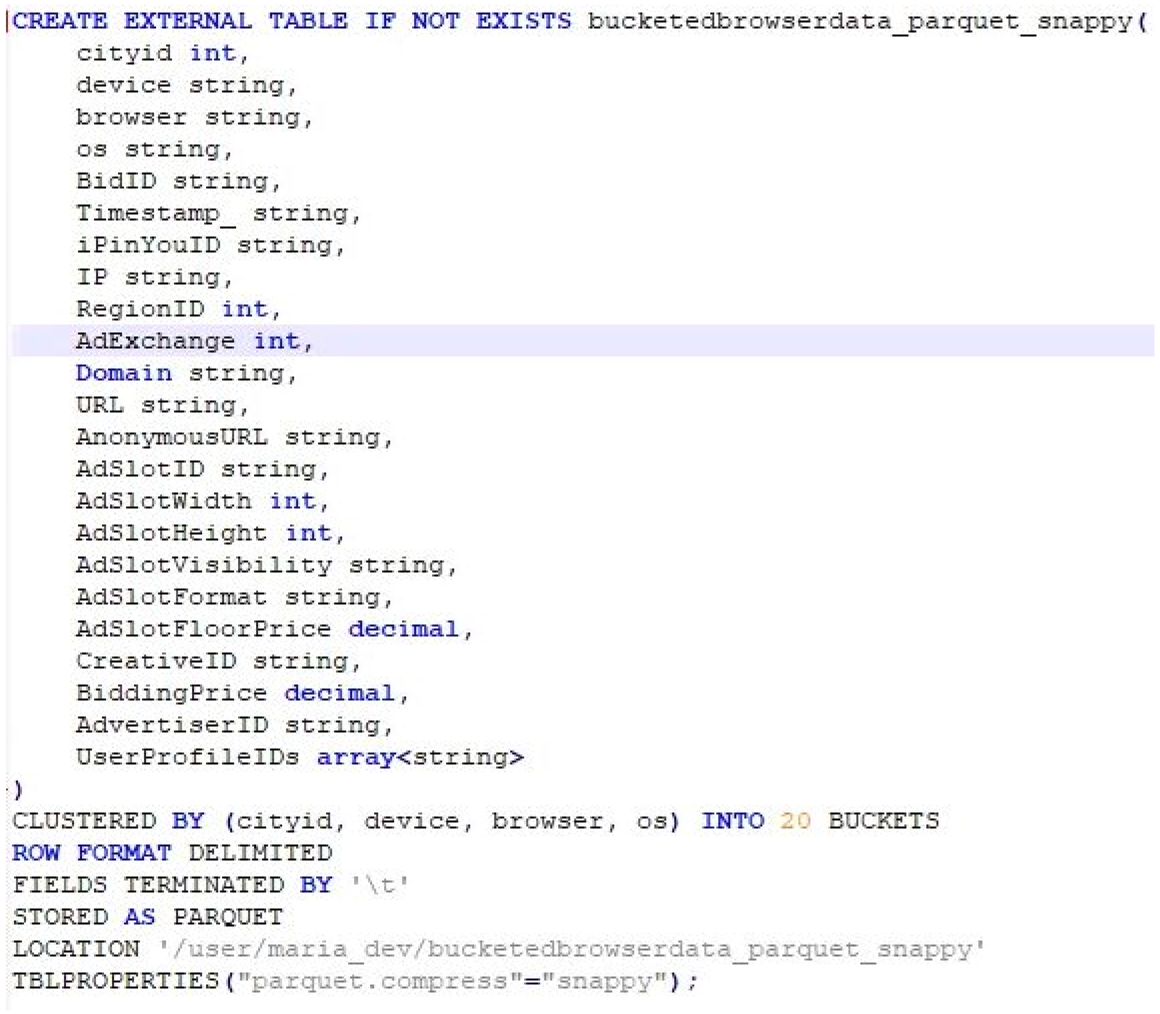

Resource Manager (CPU)
Allocated
Available
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
01:53
01:54
01:55
01:56
01:57


Resource Manager (Memory)
Allocated
Available
6GB
5GB
4GB
3GB
2GB
1GB
0MB
01:53
01:54
01:55
01:56
01:57


Disk Reads and Writes
Read
Write
22.5
20
17.5
15
12.5
10
7.5
5
2.5
0
ops
01:55
01:56
01:57
01:58
01:59

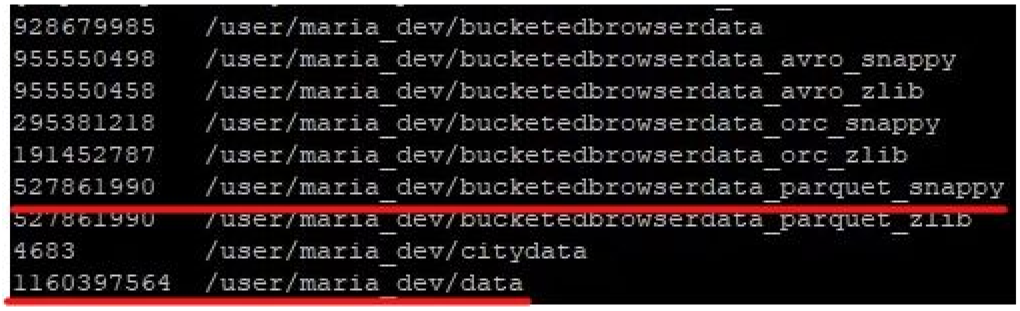
Parquet format with snappy compression.

Here’s the table definition:



```
CREATE EXTERNAL TABLE IF NOT EXISTS bucketedbrowserdata_parquet_snappy(
    cityid int,
    device string,
    browser string,
    os string,
    BidID string,
    Timestamp_ string,
    iPinYouID string,
    IP string,
    RegionID int,
    AdExchange int,
    Domain string,
    URL string,
    AnonymousURL string,
    AdSlotID string,
    AdSlotWidth int,
    AdSlotHeight int,
    AdSlotVisibility string,
    AdSlotFormat string,
    AdSlotFloorPrice decimal,
    CreativeID string,
    BiddingPrice decimal,
    AdvertiserID string,
    UserProfileIDs array<string>
)
CLUSTERED BY (cityid, device, browser, os) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS PARQUET
LOCATION '/user/maria_dev/bucketedbrowserdata_parquet_snappy'
TBLPROPERTIES("parquet.compress"="snappy");
```

And here’s the comparison between the sizes of the original and compressed tables:



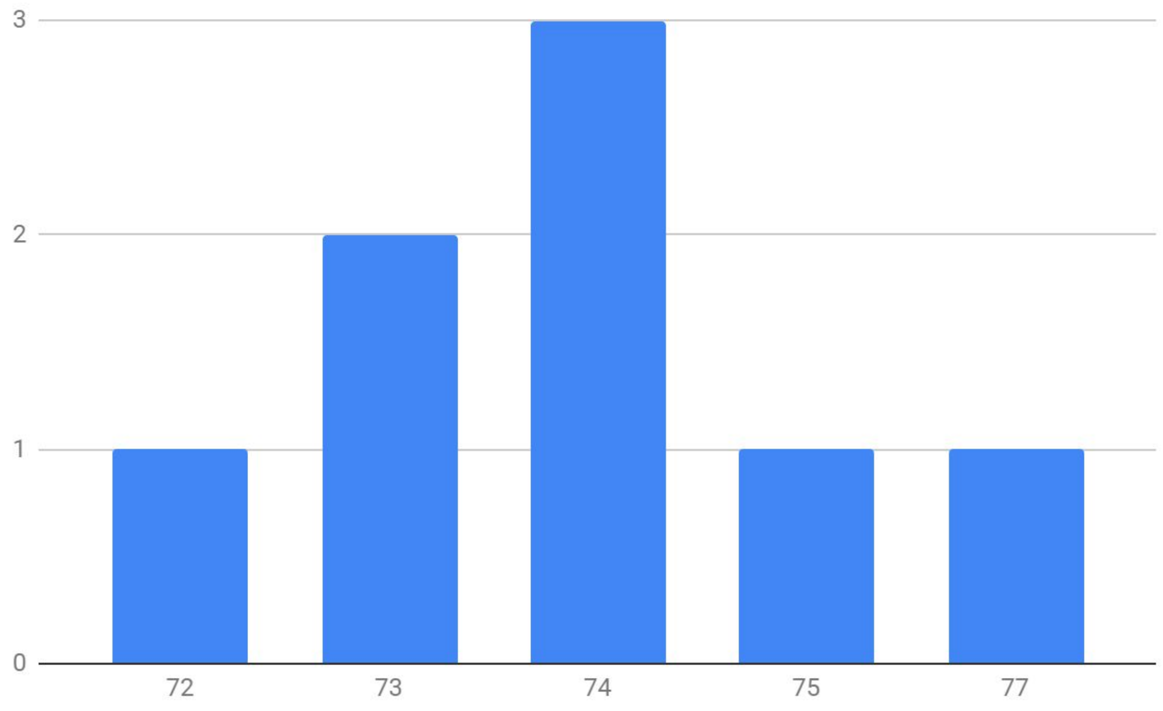
```
928679985   /user/maria_dev/bucketedbrowserdata
955550498   /user/maria_dev/bucketedbrowserdata_avro_snappy
955550458   /user/maria_dev/bucketedbrowserdata_avro_zlib
295381218   /user/maria_dev/bucketedbrowserdata_orc_snappy
191452787   /user/maria_dev/bucketedbrowserdata_orc_zlib
527861990   /user/maria_dev/bucketedbrowserdata_parquet_snappy
527861990   /user/maria_dev/bucketedbrowserdata_parquet_zlib
4683        /user/maria_dev/citydata
1160397564  /user/maria_dev/data
```

We went from 1 gig (with change) to 527 gigs.

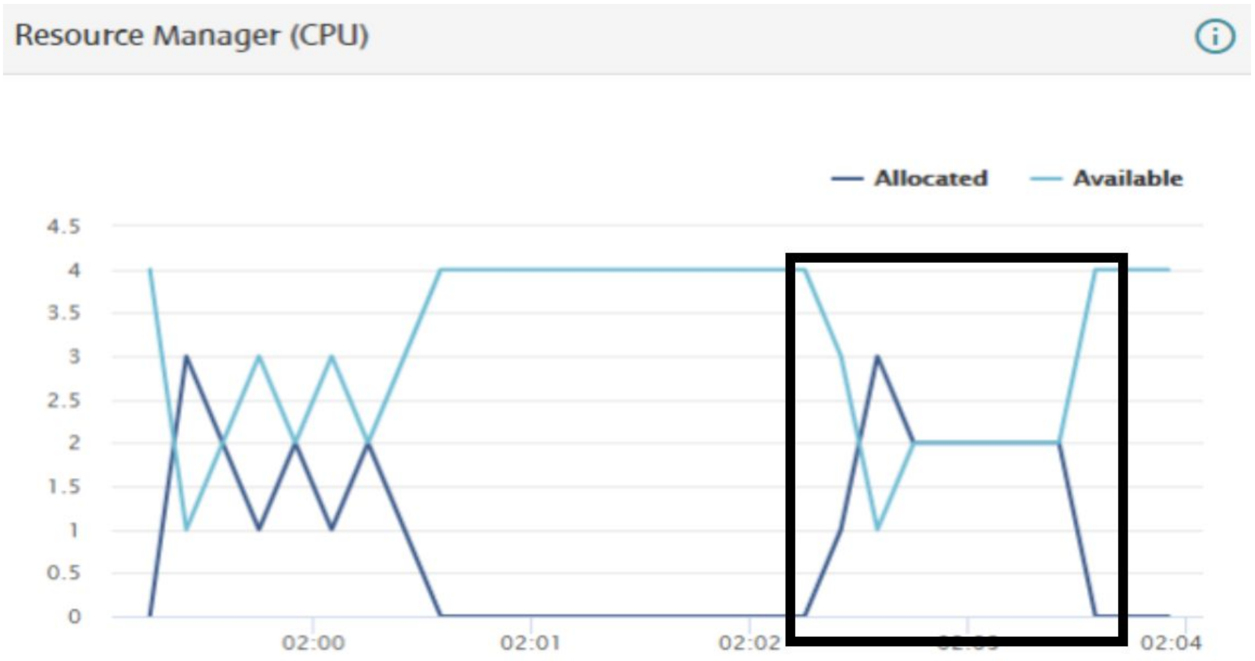
Query run results:

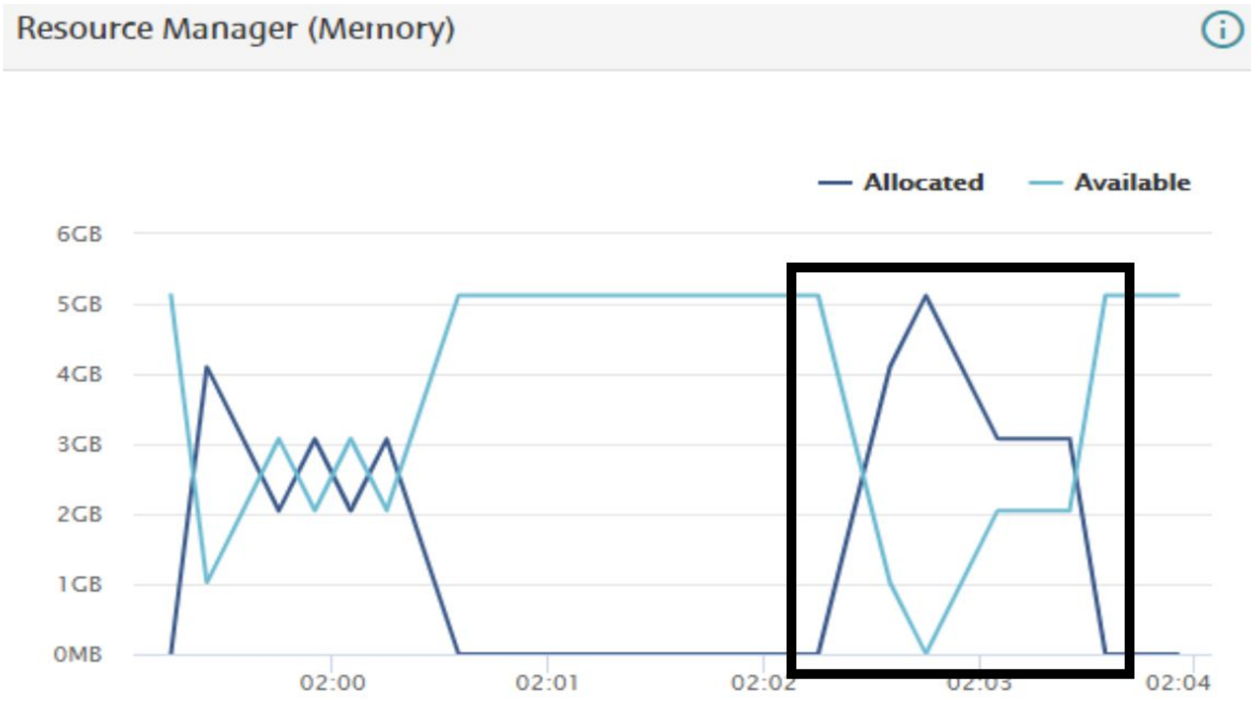
Attempt #	Run time (in seconds)

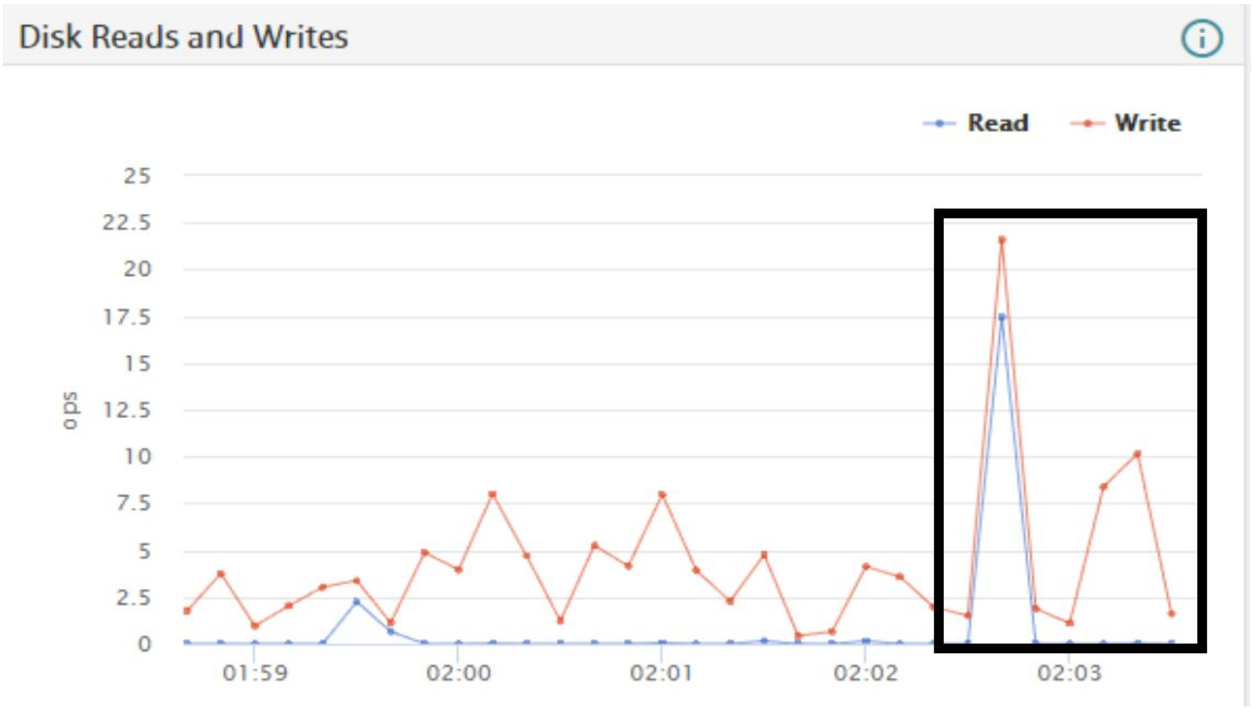
1	74
2	73
3	73
4	72
5	77
6	75
7	74
8	74
Average	**74**



It took about 215 seconds to write into this table*. RAM seemed to be the bottleneck here.*

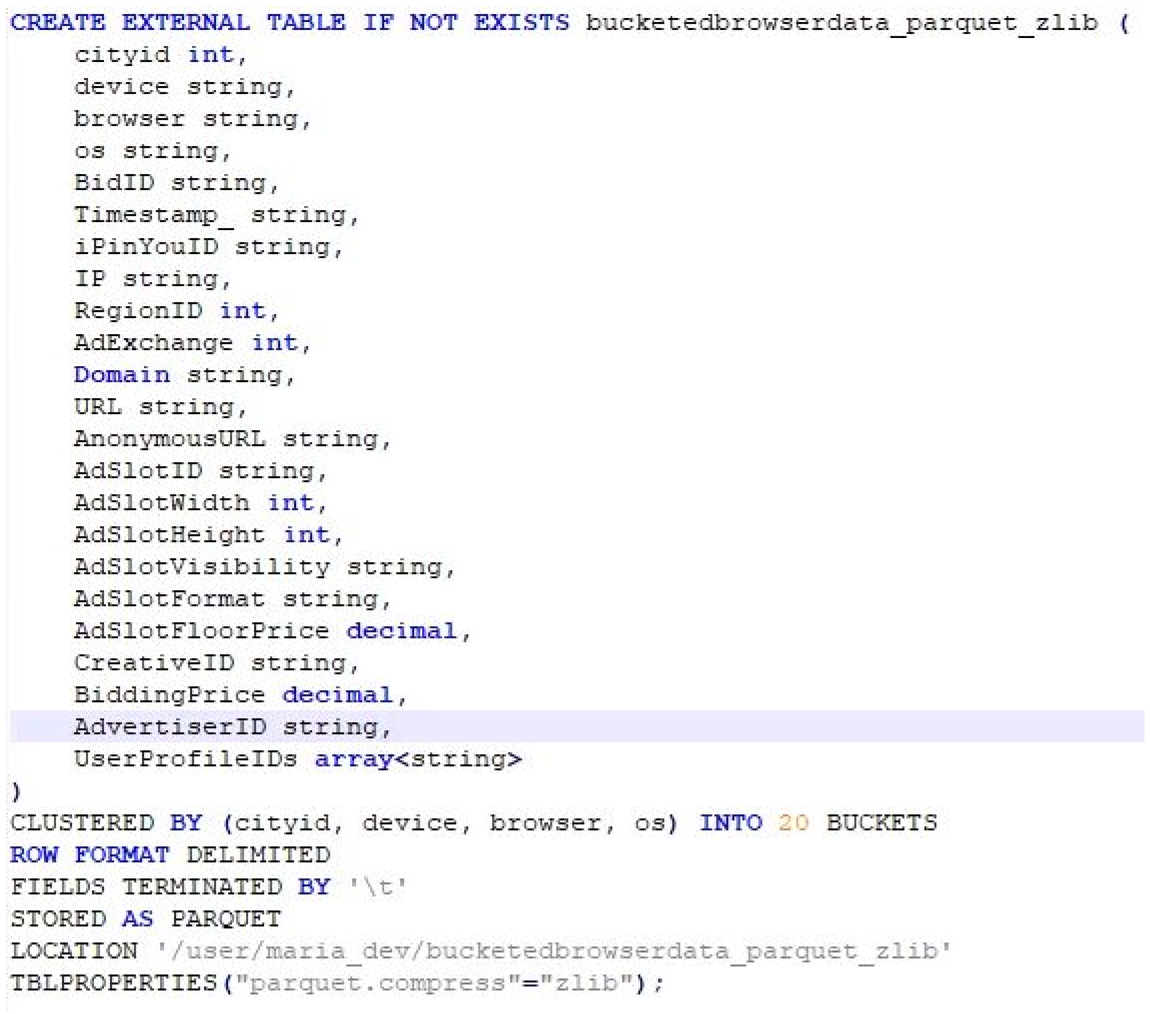

Resource Manager (CPU)
Allocated
Available
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
02:00
02:01
02:02
02:04


Resource Manager (Memory)
Allocated
Available
6GB
5GB
4GB
3GB
2GB
1GB
0MB
02:00
02:01
02:02
02:03
02:04


Disk Reads and Writes
Read
Write
25
22.5
20
17.5
15
12.5
10
7.5
5
2.5
0
ops
01:59
02:00
02:01
02:02
02:03

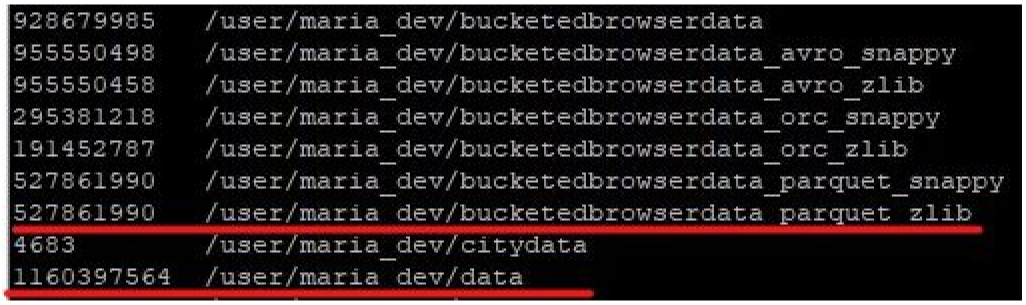
Parquet format with zlib compression.

Here's the table definition:



```
CREATE EXTERNAL TABLE IF NOT EXISTS bucketedbrowserdata_parquet_zlib (
    cityid int,
    device string,
    browser string,
    os string,
    BidID string,
    Timestamp_ string,
    iPinYouID string,
    IP string,
    RegionID int,
    AdExchange int,
    Domain string,
    URL string,
    AnonymousURL string,
    AdSlotID string,
    AdSlotWidth int,
    AdSlotHeight int,
    AdSlotVisibility string,
    AdSlotFormat string,
    AdSlotFloorPrice decimal,
    CreativeID string,
    BiddingPrice decimal,
    AdvertiserID string,
    UserProfileIDs array<string>
)
CLUSTERED BY (cityid, device, browser, os) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS PARQUET
LOCATION '/user/maria_dev/bucketedbrowserdata_parquet_zlib'
TBLPROPERTIES("parquet.compress"="zlib");
```

Here are the compression results:



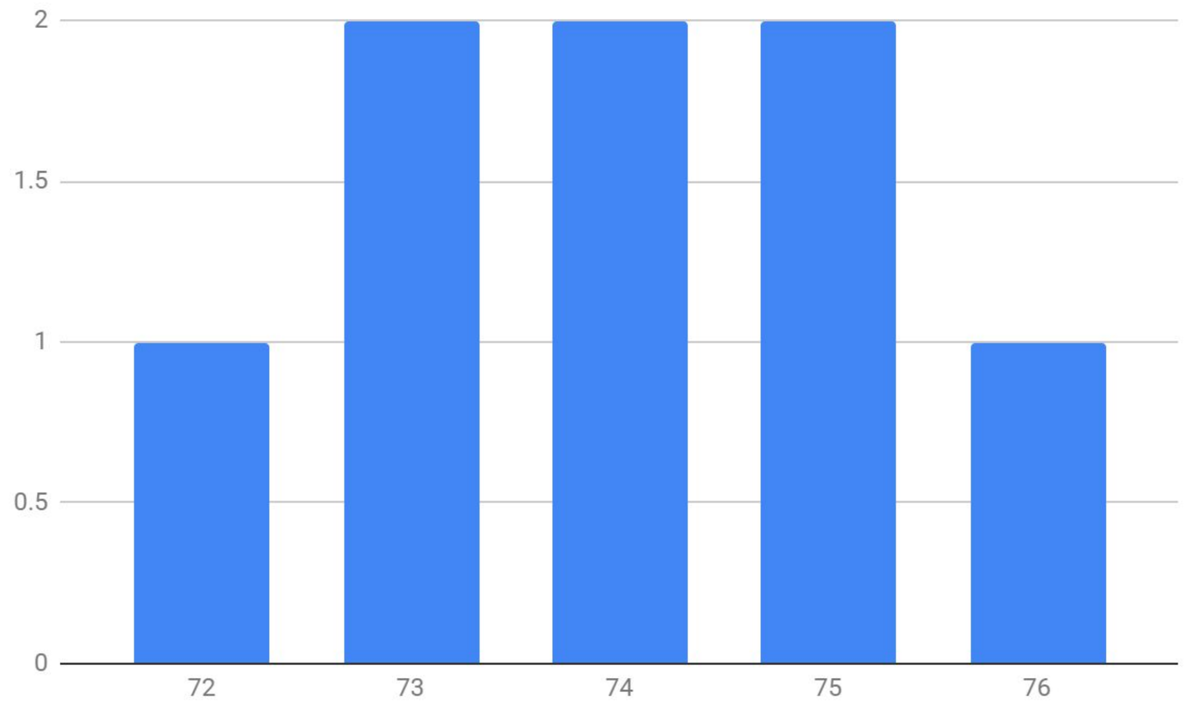
```
928679985    /user/maria_dev/bucketedbrowserdata
955550498    /user/maria_dev/bucketedbrowserdata_avro_snappy
955550458    /user/maria_dev/bucketedbrowserdata_avro_zlib
295381218    /user/maria_dev/bucketedbrowserdata_orc_snappy
191452787    /user/maria_dev/bucketedbrowserdata_orc_zlib
527861990    /user/maria_dev/bucketedbrowserdata_parquet_snappy
527861990    /user/maria_dev/bucketedbrowserdata_parquet_zlib
4683         /user/maria_dev/citydata
1160397564   /user/maria_dev/data
```

We went from 1 gig (with change) to 527 megabytes.

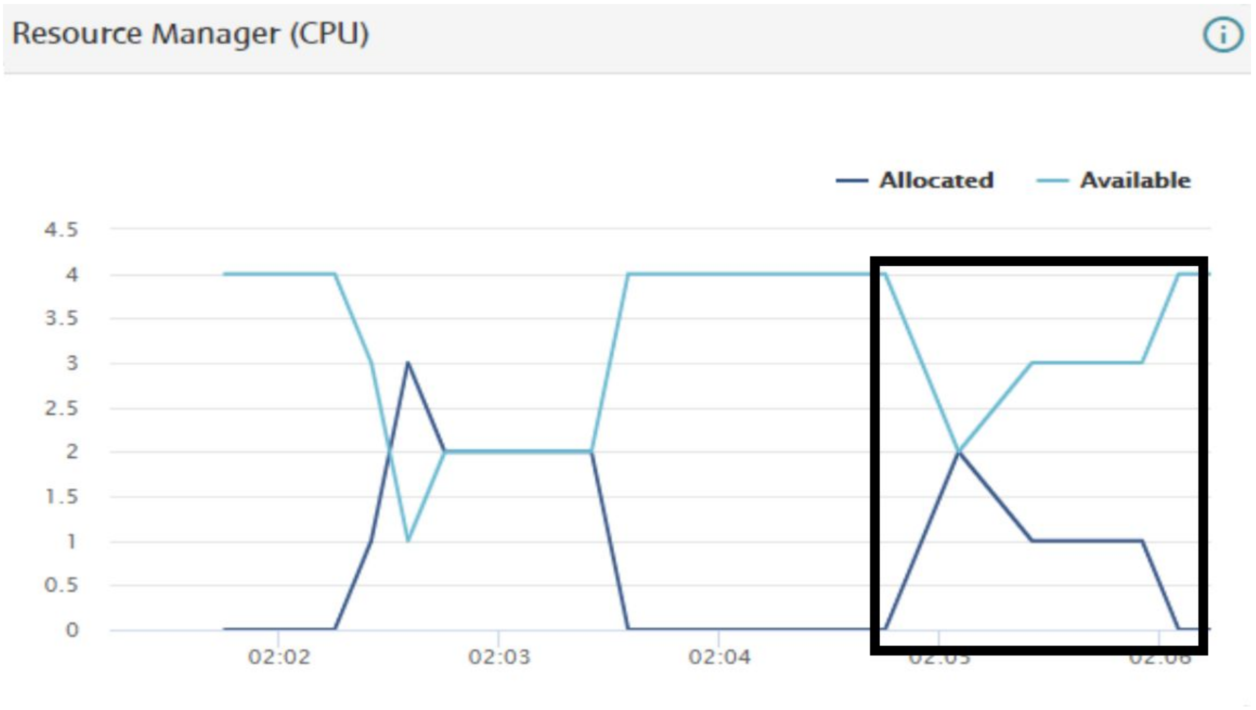
Query run results:

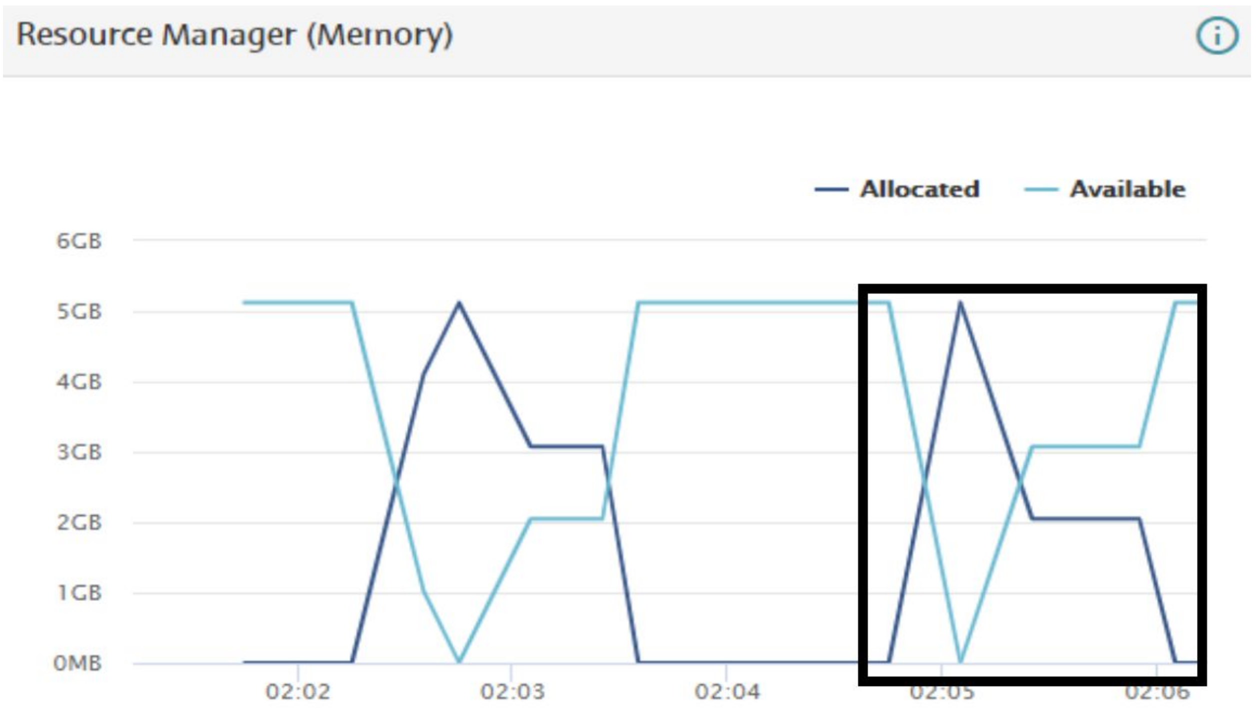
Attempt #	Run time (in seconds)

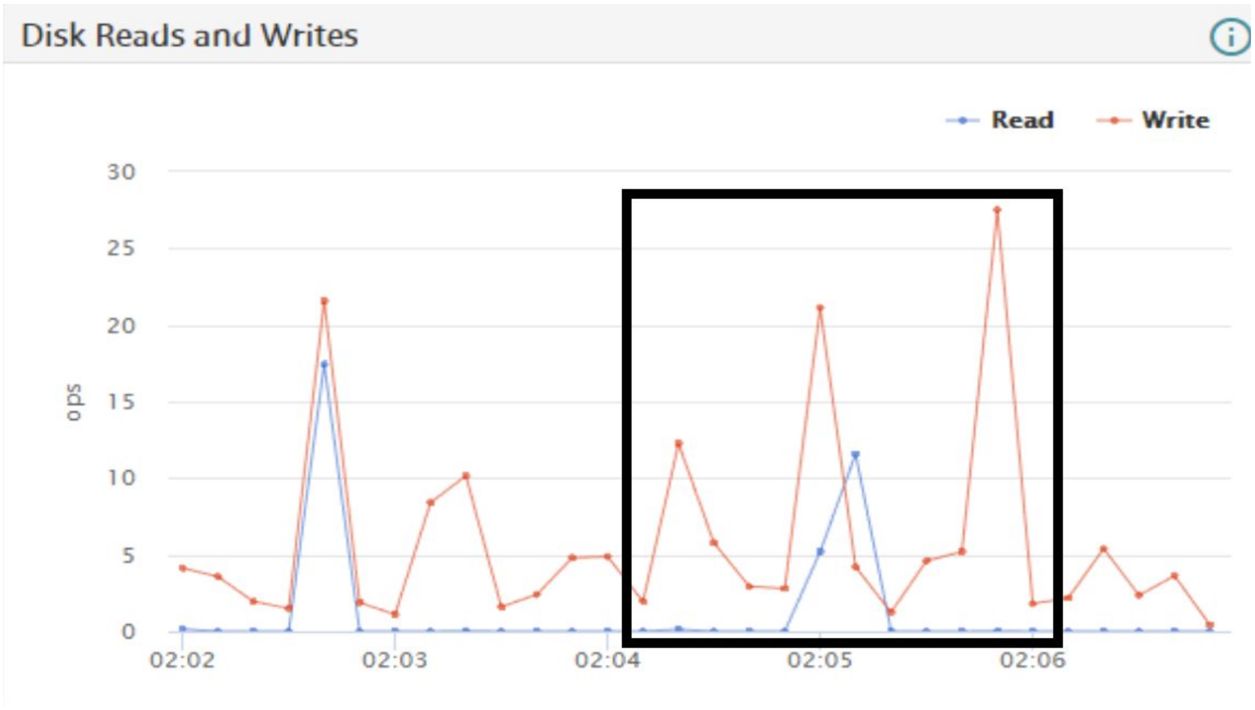
1	73
2	72
3	73
4	74
5	74
6	75
7	76
8	75
Average	**74**



It took about 201 seconds to write into this table.

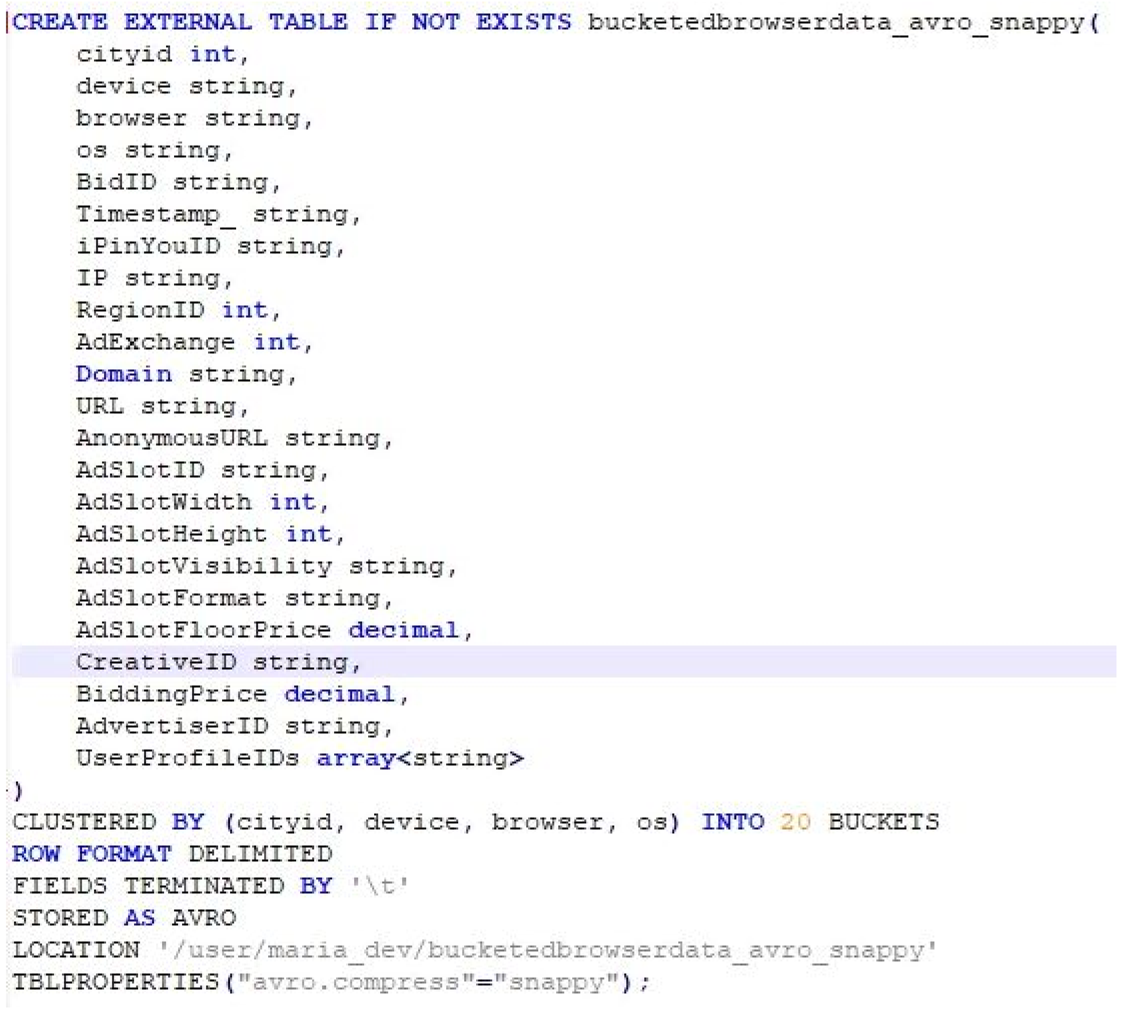

Resource Manager (CPU)
Allocated
Available
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
02:02
02:03
02:04
02:05
02:06


Resource Manager (Memory)
Allocated
Available
6GB
5GB
4GB
3GB
2GB
1GB
0MB
02:02
02:03
02:04
02:05
02:06


Disk Reads and Writes
Read
Write
ops
30
25
20
15
10
5
0
02:02
02:03
02:04
02:05
02:06

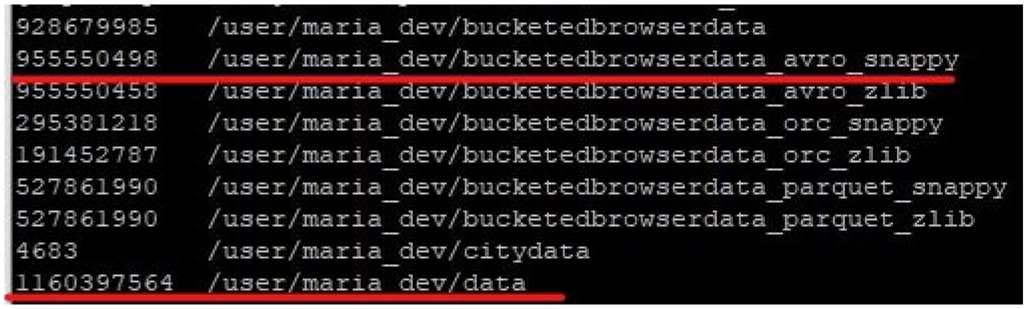
Avro format with snappy compression.

Here’s the table definition:



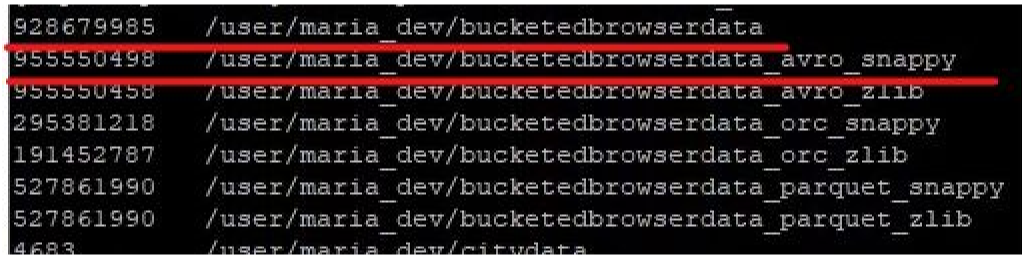
```
CREATE EXTERNAL TABLE IF NOT EXISTS bucketedbrowserdata_avro_snappy(
    cityid int,
    device string,
    browser string,
    os string,
    BidID string,
    Timestamp_ string,
    iPinYouID string,
    IP string,
    RegionID int,
    AdExchange int,
    Domain string,
    URL string,
    AnonymousURL string,
    AdSlotID string,
    AdSlotWidth int,
    AdSlotHeight int,
    AdSlotVisibility string,
    AdSlotFormat string,
    AdSlotFloorPrice decimal,
    CreativeID string,
    BiddingPrice decimal,
    AdvertiserID string,
    UserProfileIDs array<string>
)
CLUSTERED BY (cityid, device, browser, os) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS AVRO
LOCATION '/user/maria_dev/bucketedbrowserdata_avro_snappy'
TBLPROPERTIES("avro.compress"="snappy");
```

Compression results:



```
928679985    /user/maria_dev/bucketedbrowserdata
955550498    /user/maria_dev/bucketedbrowserdata_avro_snappy
955550458    /user/maria_dev/bucketedbrowserdata_avro_zlib
295381218    /user/maria_dev/bucketedbrowserdata_orc_snappy
191452787    /user/maria_dev/bucketedbrowserdata_orc_zlib
527861990    /user/maria_dev/bucketedbrowserdata_parquet_snappy
527861990    /user/maria_dev/bucketedbrowserdata_parquet_zlib
4683         /user/maria_dev/citydata
1160397564   /user/maria_dev/data
```

A very insignificant size reduction.
But we also must note this:



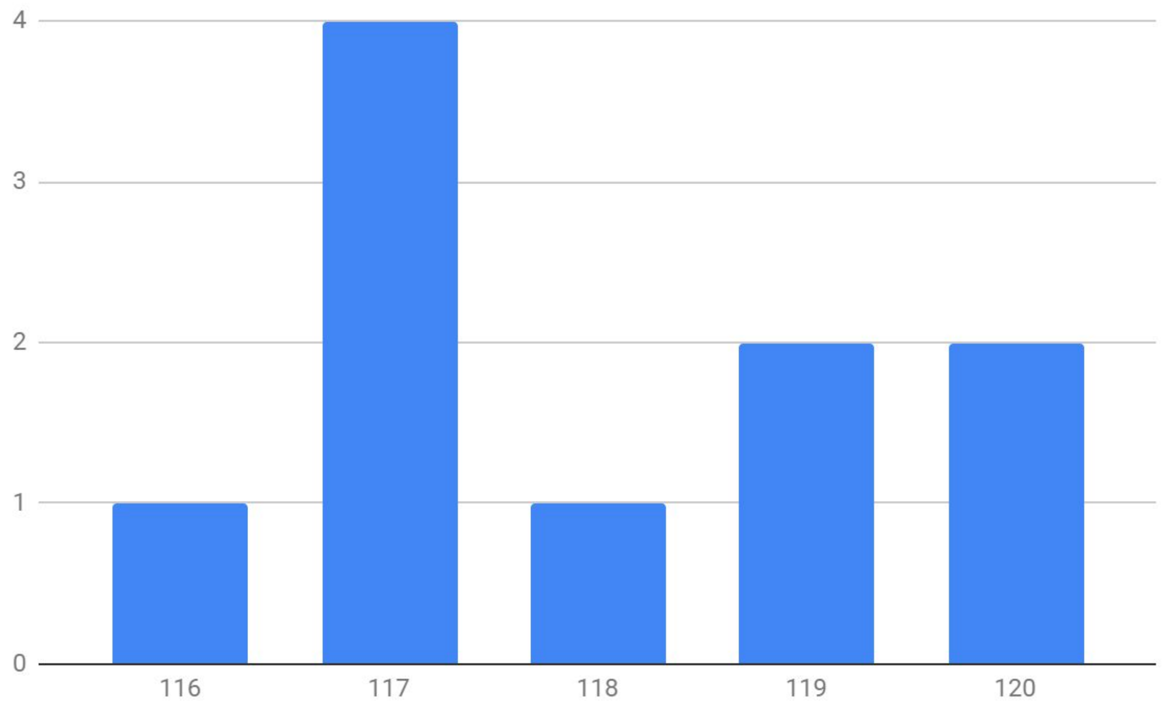
```
928679985   /user/maria_dev/bucketedbrowserdata
955550498   /user/maria_dev/bucketedbrowserdata_avro_snappy
955550458   /user/maria_dev/bucketedbrowserdata_avro_zlib
295381218   /user/maria_dev/bucketedbrowserdata_orc_snappy
191452787   /user/maria_dev/bucketedbrowserdata_orc_zlib
527861990   /user/maria_dev/bucketedbrowserdata_parquet_snappy
527861990   /user/maria_dev/bucketedbrowserdata_parquet_zlib
4683        /user/maria_dev/citydata
```

An uncompressed bucketed table is strangely smaller in size than a compressed avro table.

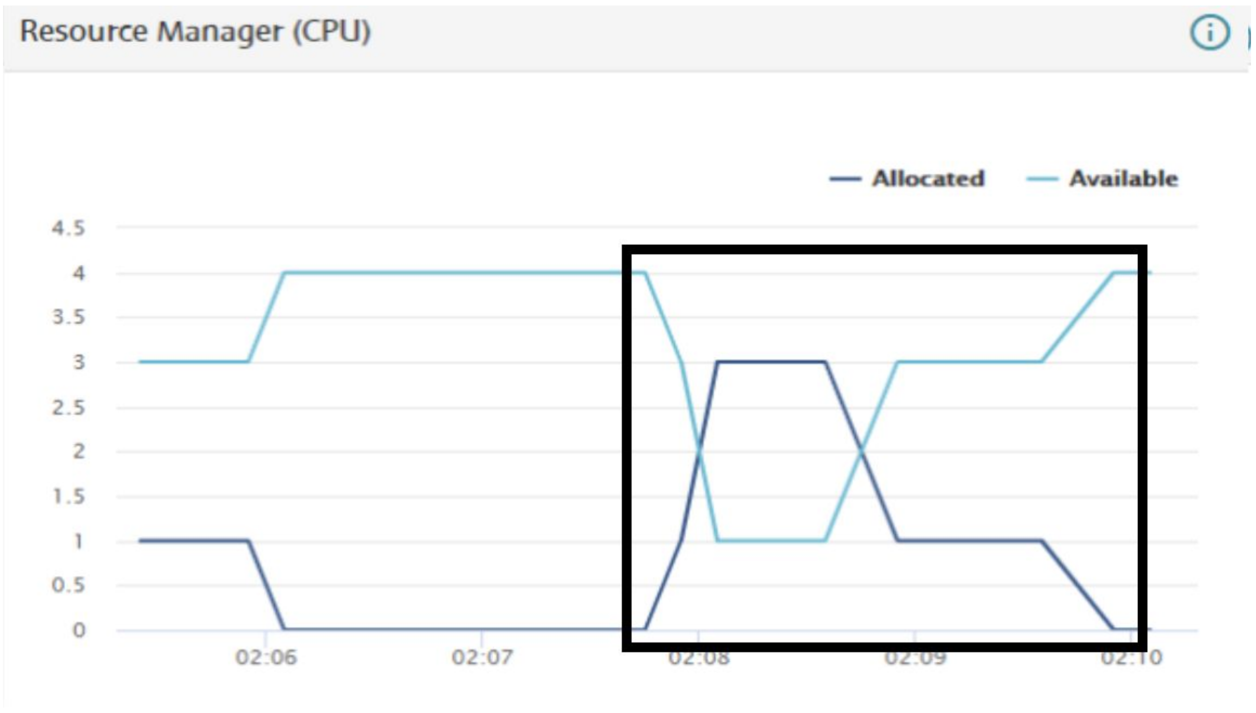
I also tried inserting from **bucketedbrowserdata** into this avro table, and the result was the same, the table simply grew in size despite using compression.

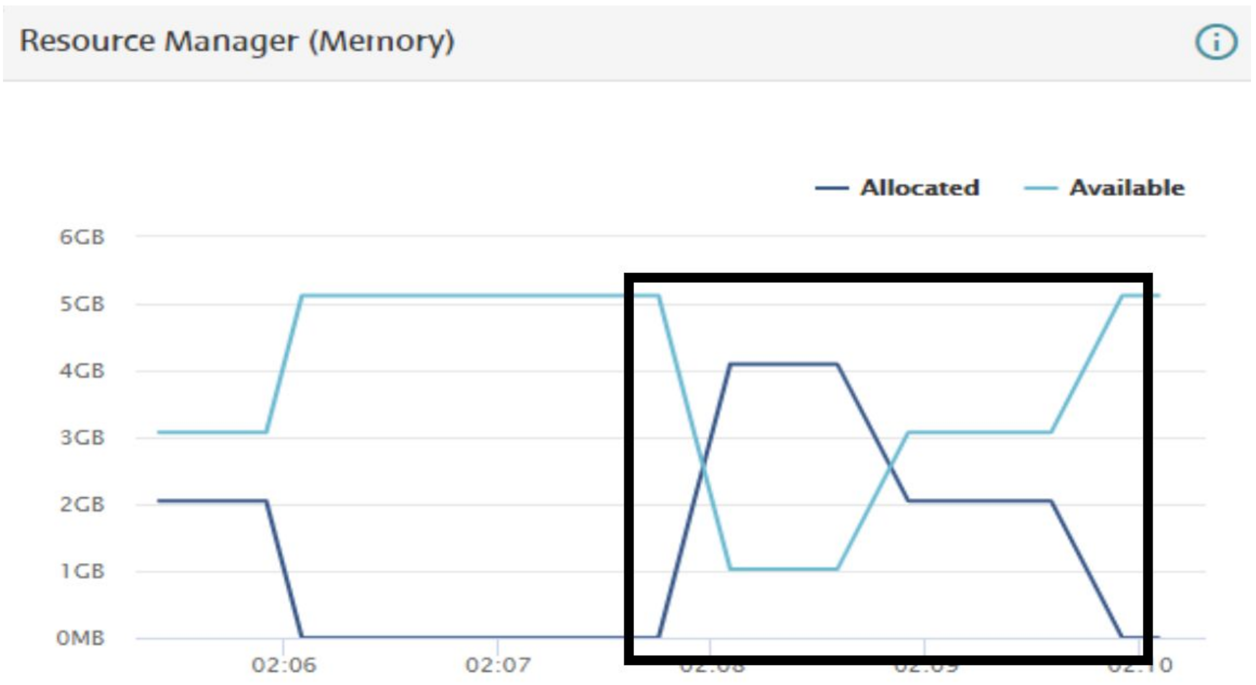
Query run results:

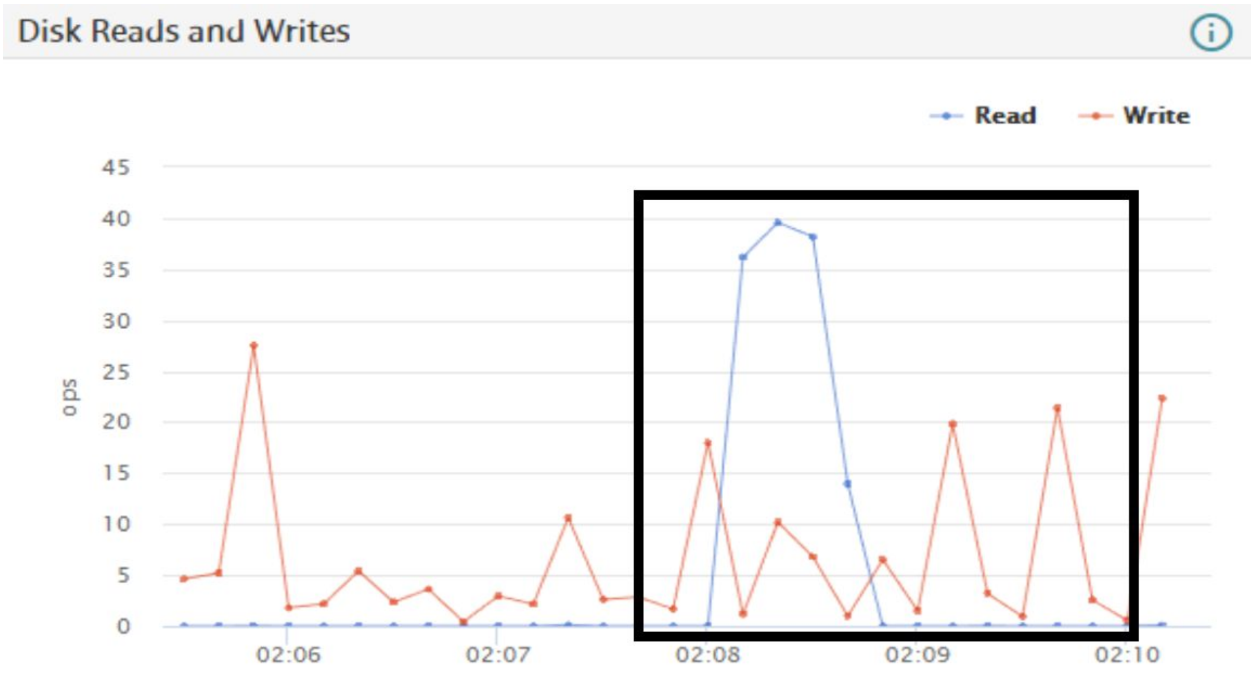
Attempt #	Run time (in seconds)
1	117
2	120
3	116
4	117
5	117
6	118
7	117
8	119
9	120
10	119
Average	**118**



It took about 213 seconds to write into this table.


Resource Manager (CPU)
Allocated
Available
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
02:06
02:07
02:08
02:09
02:10


Resource Manager (Memory)
Allocated
Available
6GB
5GB
4GB
3GB
2GB
1GB
0MB
02:06
02:07
02:08
02:09
02:10

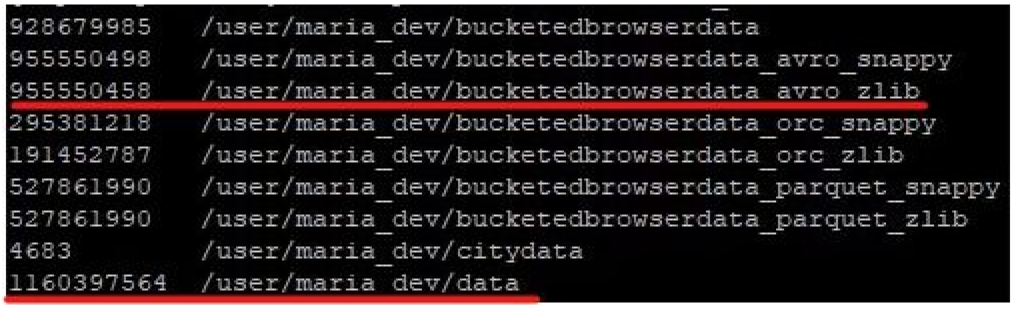

Disk Reads and Writes
Read
Write
45
40
35
30
25
20
15
10
5
0
ops
02:06
02:07
02:08
02:09
02:10

Avro format with zlib compression.

Here's the table definition:

```
CREATE EXTERNAL TABLE IF NOT EXISTS bucketedbrowserdata_avro_zlib(
    cityid int,
    device string,
    browser string,
    os string,
    BidID string,
    Timestamp_ string,
    iPinYouID string,
    IP string,
    RegionID int,
    AdExchange int,
    Domain string,
    URL string,
    AnonymousURL string,
    AdSlotID string,
    AdSlotWidth int,
    AdSlotHeight int,
    AdSlotVisibility string,
    AdSlotFormat string,
    AdSlotFloorPrice decimal,
    CreativeID string,
    BiddingPrice decimal,
    AdvertiserID string,
    UserProfileIDs array<string>
)
CLUSTERED BY (cityid, device, browser, os) INTO 20 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS AVRO
LOCATION '/user/maria_dev/bucketedbrowserdata_avro_zlib'
TBLPROPERTIES("zlib.compress"="zlib");
```

Compression results:



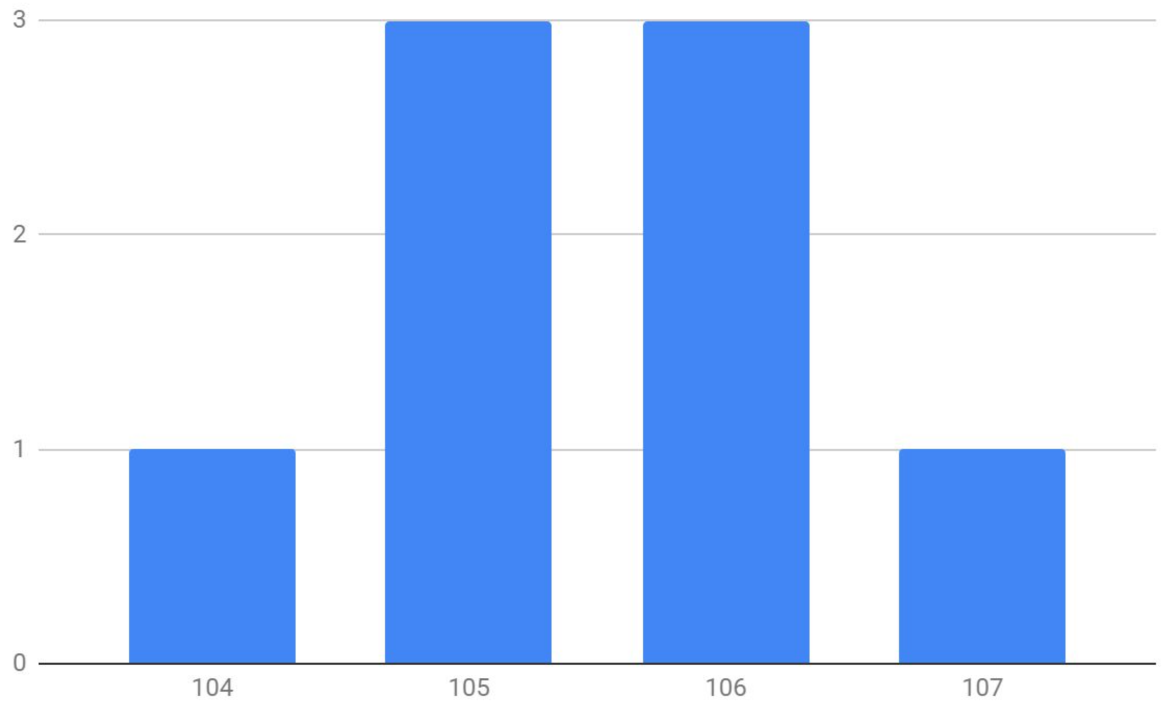
```
928679985   /user/maria_dev/bucketedbrowserdata
955550498   /user/maria_dev/bucketedbrowserdata_avro_snappy
955550458   /user/maria_dev/bucketedbrowserdata_avro_zlib
295381218   /user/maria_dev/bucketedbrowserdata_orc_snappy
191452787   /user/maria_dev/bucketedbrowserdata_orc_zlib
527861990   /user/maria_dev/bucketedbrowserdata_parquet_snappy
527861990   /user/maria_dev/bucketedbrowserdata_parquet_zlib
4683        /user/maria_dev/citydata
1160397564  /user/maria_dev/data
```

The compression results are once again as strange as in the previous paragraph.

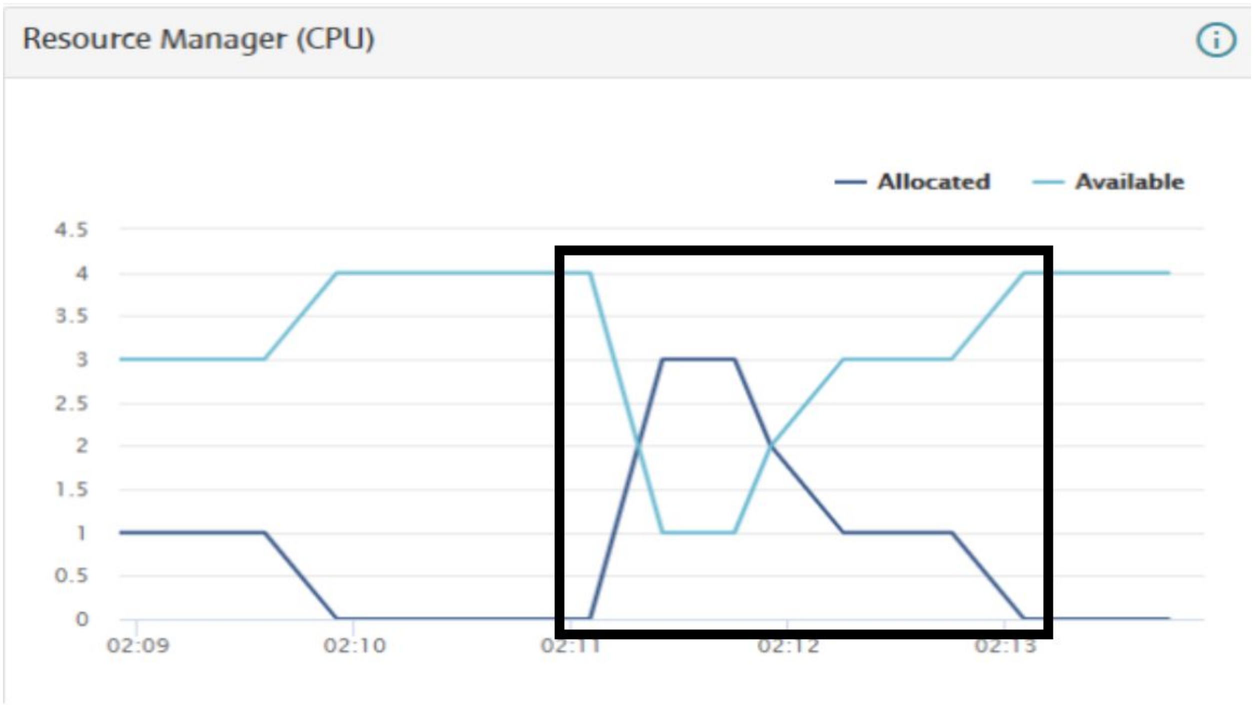
Query run results:

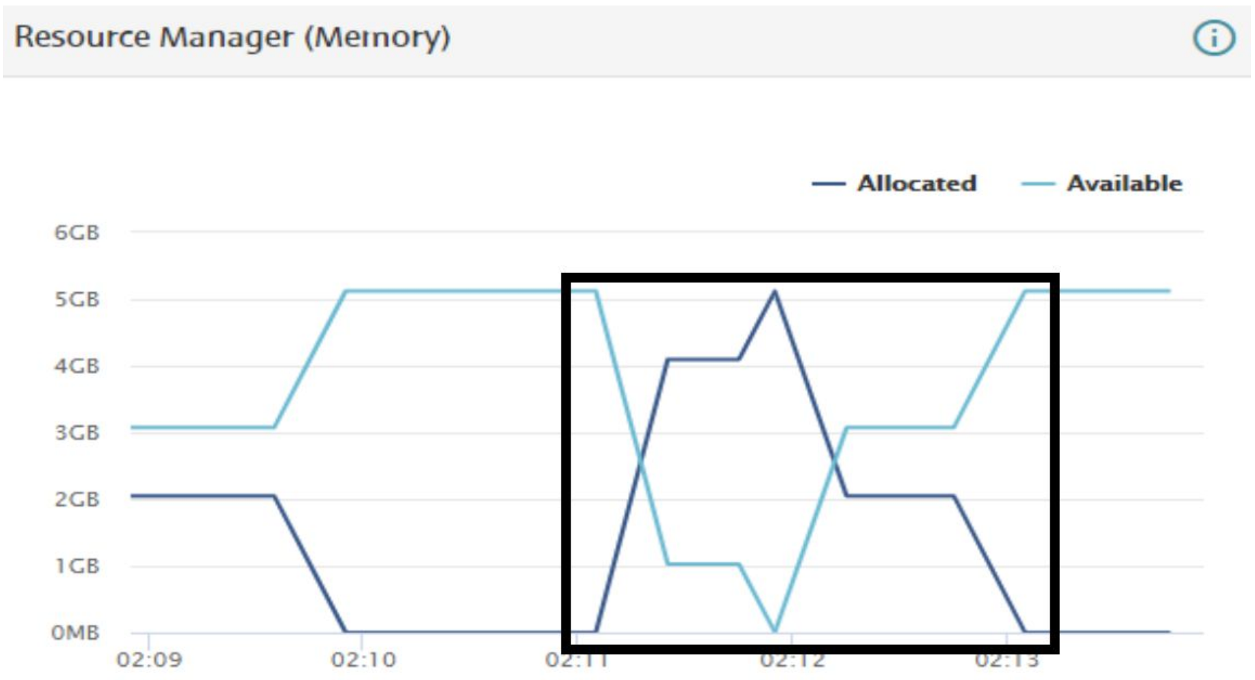
Attempt #	Run time (in seconds)

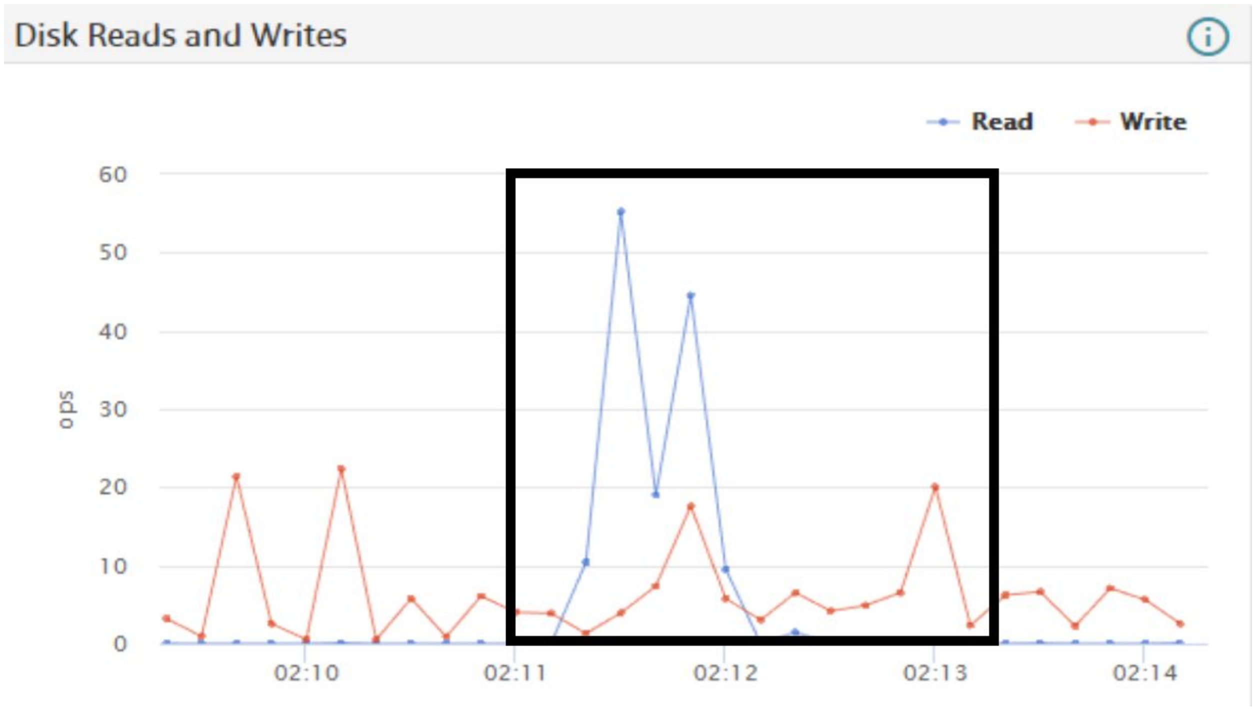
1	105
2	104
3	105
4	106
5	107
6	106
7	105
8	106
Average	**105,5**



It took about 212 seconds to write into this table.


Resource Manager (CPU)
Allocated
Available
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
02:09
02:10
02:11
02:12
02:13


Resource Manager (Memory)
Allocated
Available
6GB
5GB
4GB
3GB
2GB
1GB
0MB
02:09
02:10
02:11
02:12
02:13

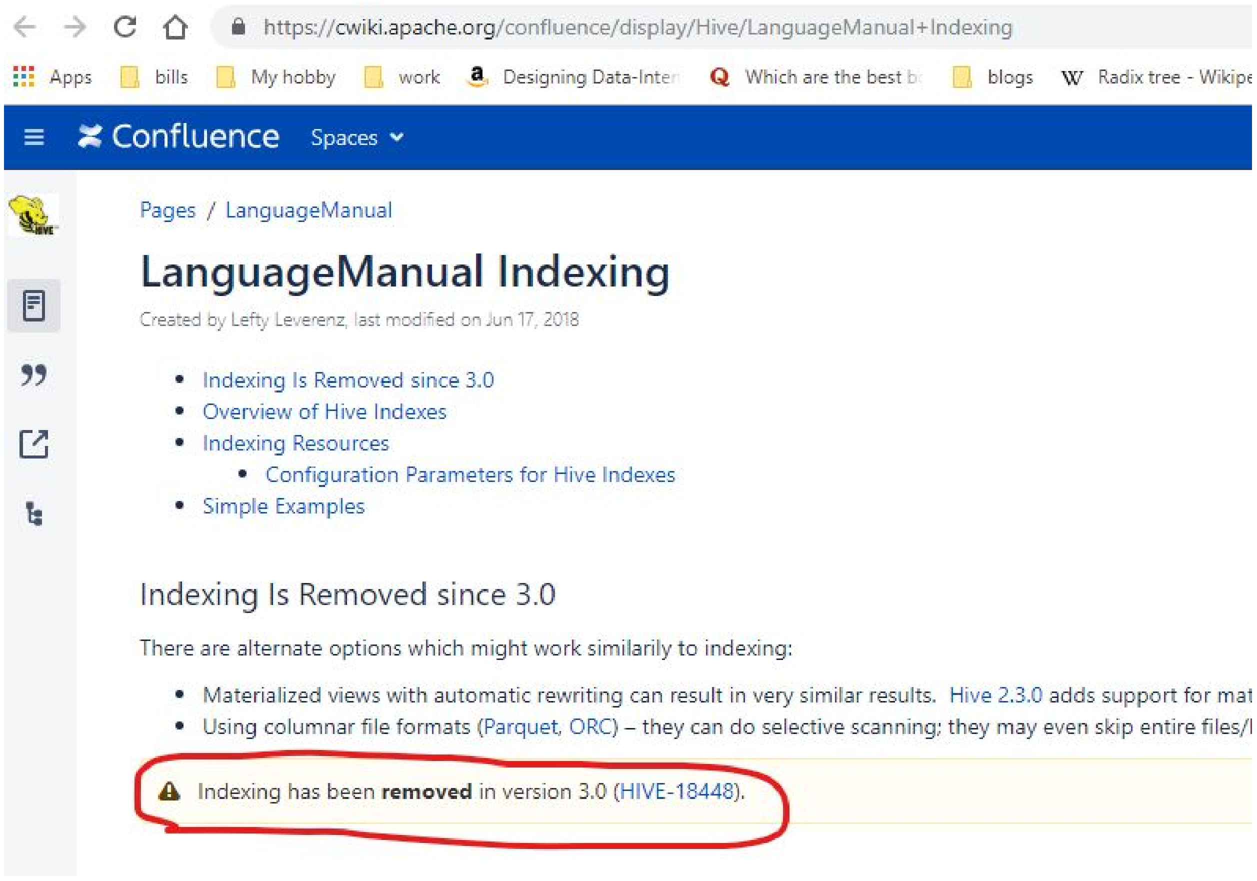


Conclusion.

ORC with zlib is the best compression method. It offers built in indexes and significantly reduces the size of our data.

Indexing.

Official statement from hive documentation:


https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Indexing

Pages / LanguageManual

LanguageManual Indexing

Created by Lefty Leverenz, last modified on Jun 17, 2018

- Indexing Is Removed since 3.0
- Overview of Hive Indexes
- Indexing Resources
  - Configuration Parameters for Hive Indexes
- Simple Examples

Indexing Is Removed since 3.0

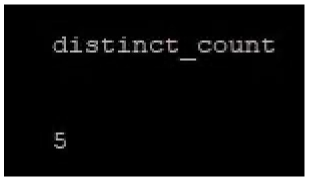
There are alternate options which might work similarly to indexing:

- Materialized views with automatic rewriting can result in very similar results. Hive 2.3.0 adds support for mat
- Using columnar file formats (Parquet, ORC) – they can do selective scanning; they may even skip entire files/l

Indexing has been **removed** in version 3.0 (HIVE-18448).

As you can see, the doc suggests that we should rely on indexes in Parquet and ORC instead.

But anyways, let's take our **brucketedbrowserdata** table and see how it performs with indexing. Bitmap indexing is supposed to be used only when there are a few columns. In our case, only the **device** column fits the bill:



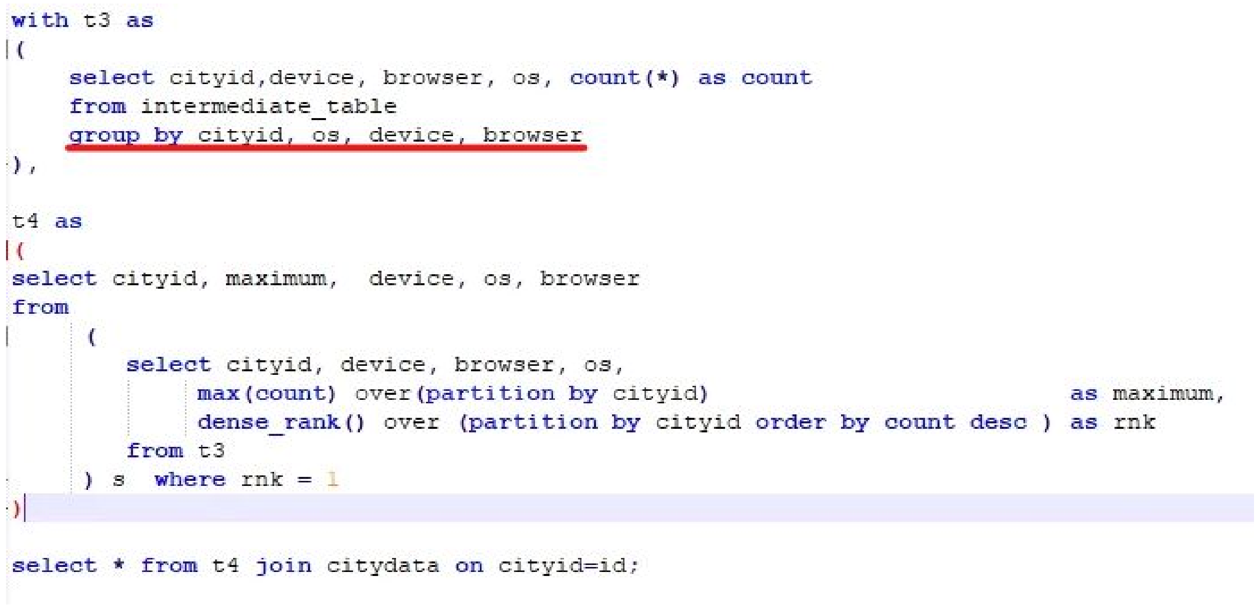
```
distinct_count

5
```

Supposedly, indexing is going to improve group by performance. Let's add an index on the device column:

```
CREATE INDEX device_index
ON TABLE bucketedbrowserdata (device)
AS 'BITMAP'
WITH DEFERRED REBUILD;
```

This didn't improve the performance at all, so I am not going to include the run results. The run time of the query is the same with and without this index.

Let's try to add a compact index on the columns that we have in group by:



```
with t3 as
(
    select cityid,device, browser, os, count(*) as count
    from intermediate_table
    group by cityid, os, device, browser
),

t4 as
(
select cityid, maximum,  device, os, browser
from
        (
            select cityid, device, browser, os,
                max(count) over(partition by cityid)                          as maximum,
                dense_rank() over (partition by cityid order by count desc )  as rnk
            from t3
        ) s  where rnk = 1
)

select * from t4 join citydata on cityid=id;
```

This didn't improve anything either bucketed browserdata performs the same with and without this index.

Conclusion.

The removal of indexes from hive is certainly warranted, it seems that they don't help much at all.

Vectorization.

Enabling vectorization did not help that much at all. The results are the same as the results without it, so I did not bother including them at all.

Tez vs MapReduce.

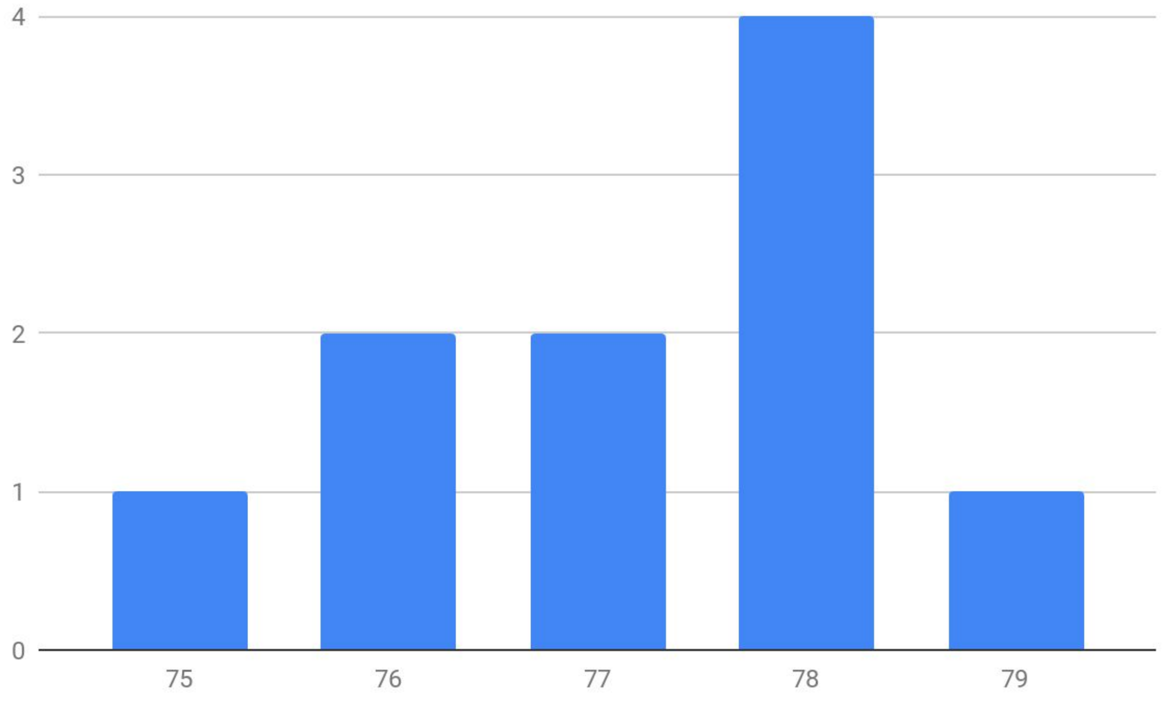
Alas, the version of hive that comes with MapR 6.1 (hive version 2.3.3.) has a bug. This bug is such that whenever you have a window function in your query, it may only work in a new container, therefore, whenever you run your query with window functions, if tez reuses an old container, it fails and starts a new one, which we all know is a bad thing, because starting a new container takes time, and this, of course, degrades performance dramatically. In my case the performance was degraded to the point where mapreduce queries were faster than tez queries. Click here to learn more about this bug.

All this made me do my last task on hortonworks…

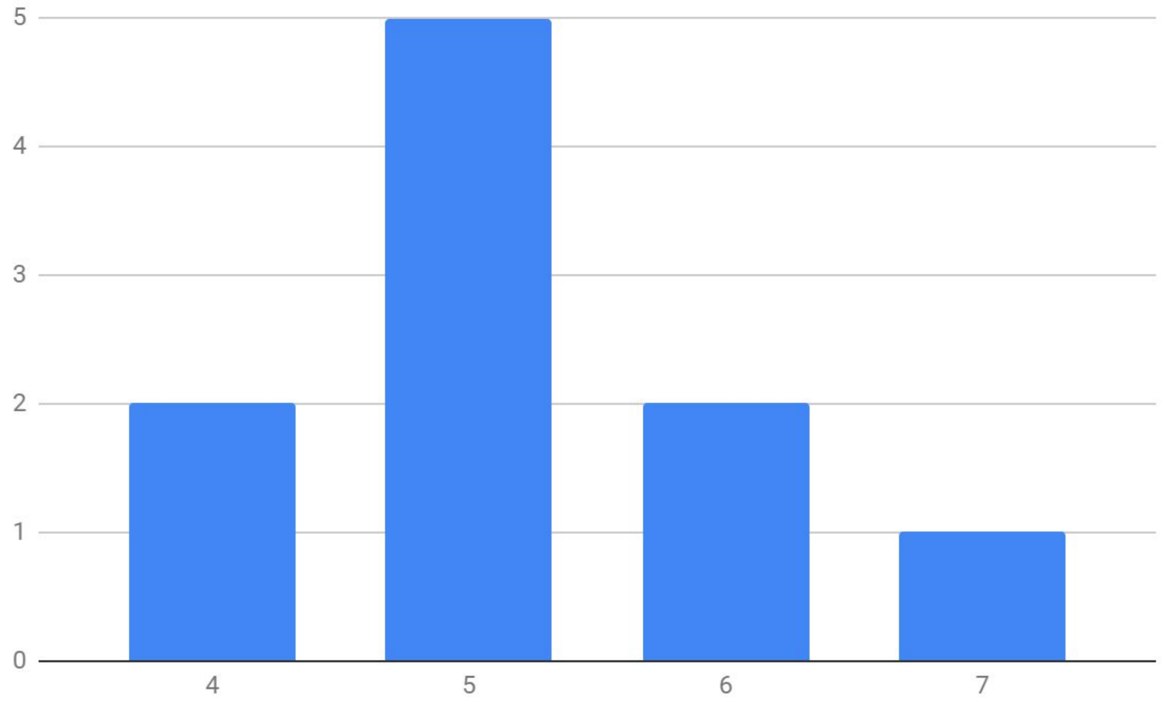
Mapreduce vs tez:

Attempt #	Map reduce run time (in seconds)	Tez run time (in seconds)
1	78	4
2	79	4
3	78	7
4	78	5
5	78	5
6	76	5
7	76	6
8	75	6
9	77	5
10	77	5
Average	**77,2**	**5,2**

Map reduce distribution:

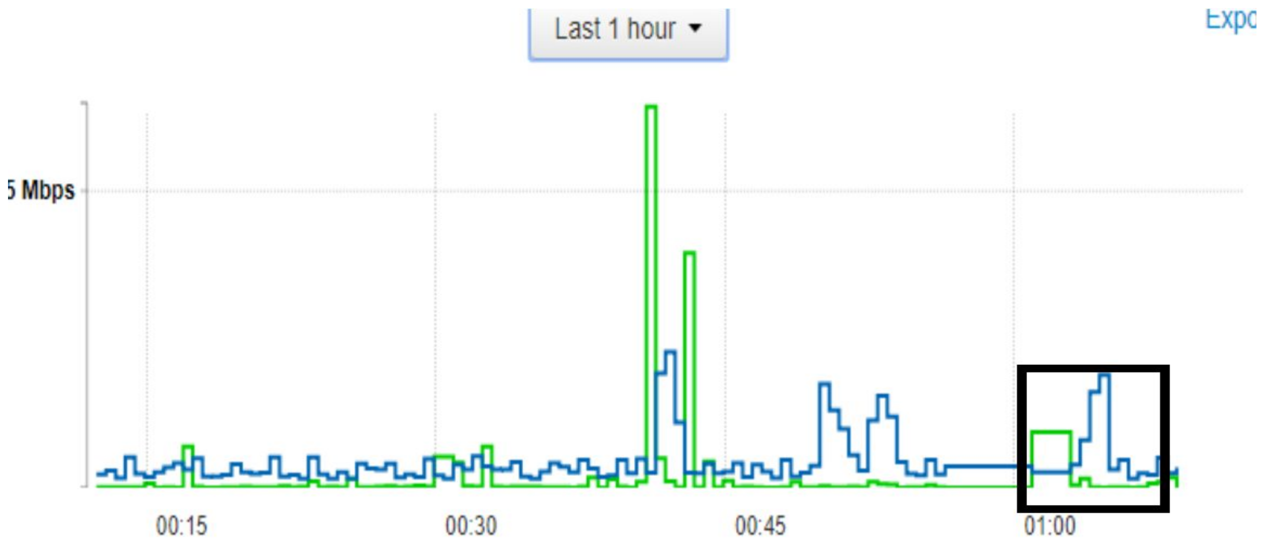


Tez distribution:

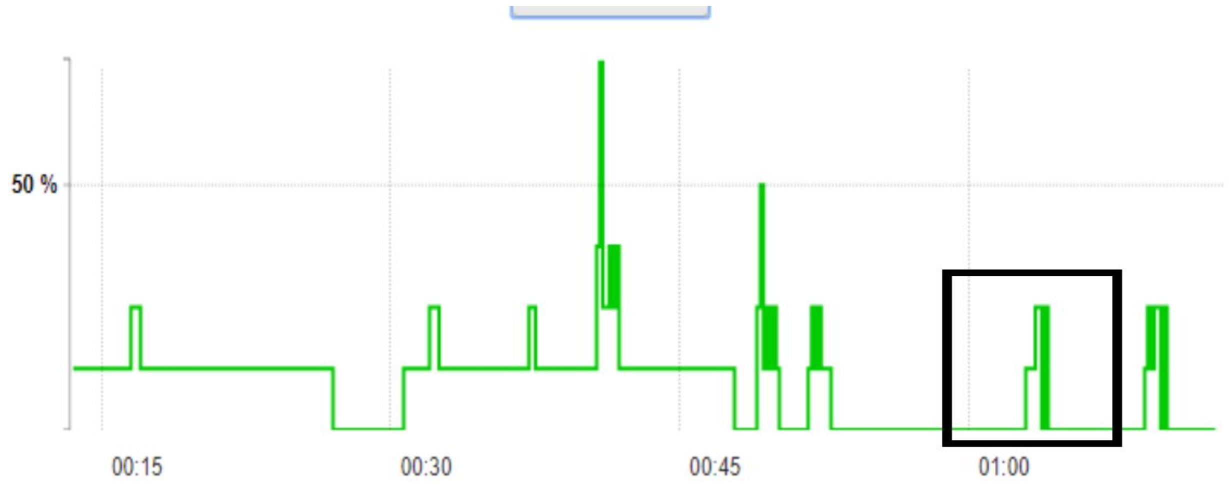


Mapreduce metrics:

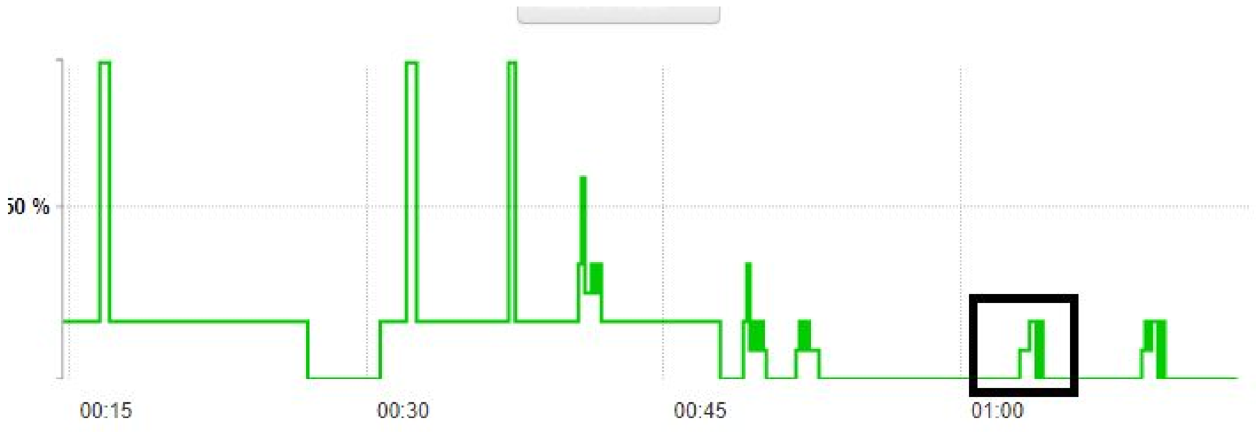
Disk:



The CPU was pretty idle.

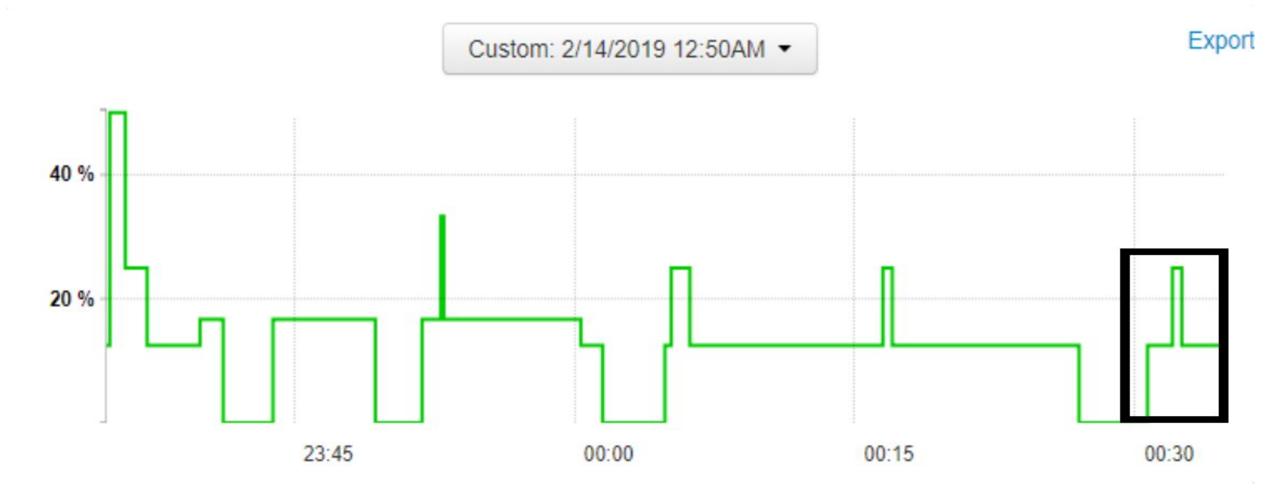


Memory:

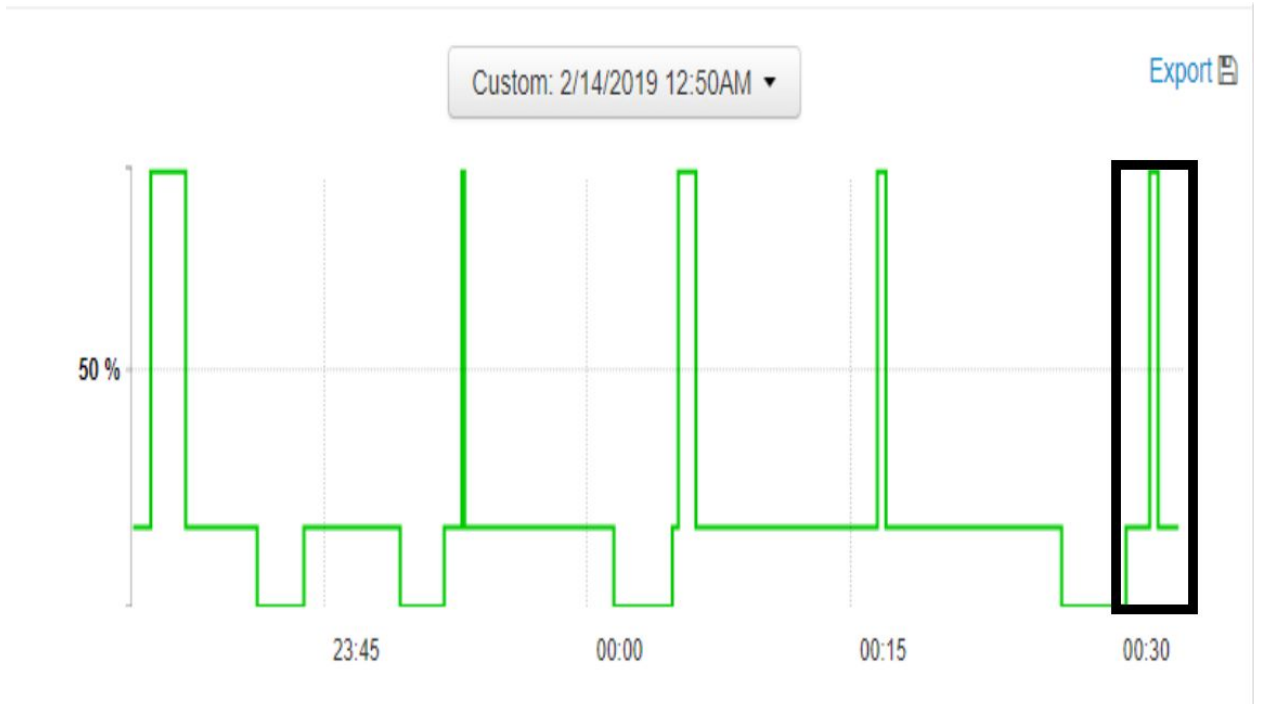


Here are the results for tez:

CPU:



RAM:



It seems that RAM is the main bottleneck here. The disk usage was extremely insignificant, so I didn't add a screenshot.

Conclusion.

Tez improves performance dramatically over map reduce, in fact hive version 3.1.0 on hortonworks version 3.0.0 doesn't even include map reduce anymore. So, having a lot of ram will definitely speed up your queries.

Tez is faster than map reduce because it stores the intermediate results of DAG actions in memory whereas map reduce reads/writes intermediate results of mappers, combiners and reducers from/to disk.

Final conclusion.

The best schema of all turned out to be the one bucketed by **cityid**, **os**, **device** and **browser**.

The best compression is ORC with zlib, because it offers the smallest size and the performance is the same as with the snappy compression.

And running your queries is, of course, much better on tez than on mapreduce. Here's a summary table comprising all of my results:

Schema	Average runtime (in seconds)	File size (in gigabytes)
Default	90,5	1,2
Partitioned by cityid	90,125	1,2
Bucketed by cityid	85,75	1,2
Bucketed by cityid,device, os, browser	79,(09)	1,2
ORC with snappy	66,125	0,295
ORC with zlib	64,875	0,191
Parquet with snappy	74	0,527
Parquet with zlib	74	0,527
Avro with snappy compression	118	0,955
Avro with zlib compression	105,5	0,955

The best schema from above is ORC with zlib. Here's ORC with zlib on mapreduce vs tez:

Engine	Average runtime (in seconds)
Mapreduce	77,2
Tez	5,2