Timestamp Resistant Query Caching in Append-Only Databases

Draft: September 27, 2018

ABSTRACT

We demonstrate a method for caching and re-using intermediate query results for simple parallel OLAP aggregations on immutable data that is amenable to the dashboard usage pattern by using a per block query cache to store query results.

Significantly, our method is mildly robust to changing time predicates, allowing partial results to be shared between queries with different time filters.

The practical importance of our method is that queries that re-evaluate the same data are able to achieve significant (2x - 10x) speed up by avoiding redundant computations while still showing up to date query results.

Keywords

OLAP, map-reduce, append-only data, row-block, columnar storage, chronological data

1. INTRODUCTION

During the day to day operations of engineers and managers in large organizations, they make use of dashboards [2] that show them the health, status and other vital metrics of their systems. Each team may have multiple dashboards, each with dozens of graphs on it.

The dashboard usage pattern causes a burst of queries which stresses the underlying databases. One solution is to cache the query results at the UI layer and only query the backends once every minute or hour, leaving the clients with stale data. Another solution is to use datastores that have pre-aggregated results for these queries, like a Time Series Database (TSDB).

We demonstrate a method for caching and re-using query results for simple Online Analytical Processing (OLAP) queries that is amenable to the dashboard usage pattern by using a per block result cache to store intermediate query results instead of caching the final result.

The major difficulty with a query result cache is that most dashboard queries use continually changing time predicates.

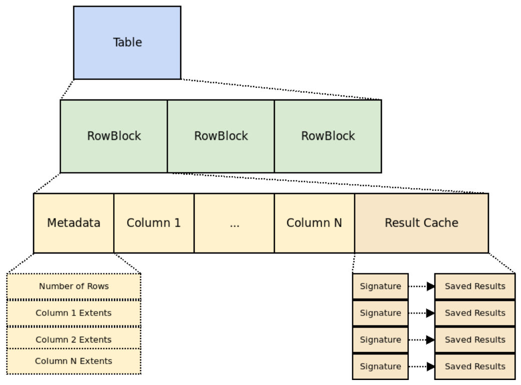


Figure 1: Table and RowBlock Layout with RowBlock result cache

A query executed now and a query executed in 5 seconds from now will have different time predicates, causing cache misses. To remediate, we use a method similar to rowblock elimination [2] on the filters for the query on a block by block basis, removing any filters that are true for all records in a block.

Because most data arrives naturally in chronological order, this method works well for time filters; any blocks that are wholly contained within the query time range will have the time range filters removed from their cache signature and are able to re-use computations from previous queries.

Thus, with per block query result caching and filter elimination, new computation is only performed on partially relevant blocks and combined with cached results from previously computed blocks.

2. IMPLEMENTATION

2.1 Architecture

The method presented is implemented in Sybil - an open source[1] multi threaded execution engine on append-only column-oriented data. Sybil is inspired by the Scuba engine[2] and like Scuba, data in Sybil is stored in RowBlocks of 65K records. For each column in a block, the min and max values are maintained as metadata. (Figure 1)

Like Scuba, Sybil's execution engine is a parallel query engine that supports simple aggregations without JOINS. For each query, there is a threaded worker pool and each worker aggregates a block into its ResultSet. A combining thread takes the ResultSets and merges them together. This con-

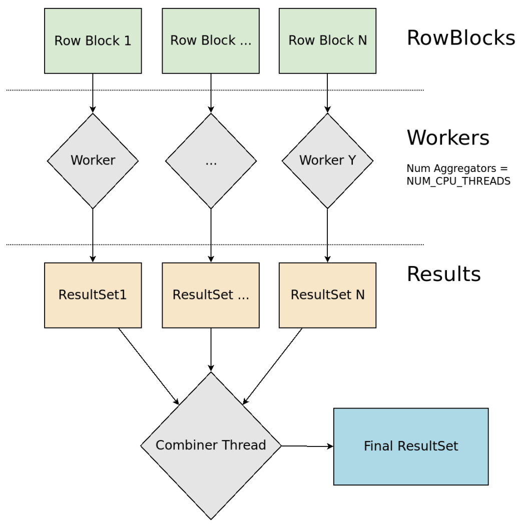


Figure 2: Parallel Aggregation Pattern

tinues until there is only one ResultSet remaining. (Figure 2) This process can happen on one machine or across many.

2.2 Result Cache

To add query result caching to Sybil, we add a new step when aggregating the rows in a block. Before loading and querying the data in a block, we hash the query spec into a signature and check to see if the signature exists in that block's query cache. For any block with cache hits, we then pull the cached ResultSet off disk and avoid loading the records for this block off disk.

For blocks with cache misses, if the block is at its maximum capacity (and is now immutable), the ResultSet is saved to that block's query cache for usage in subsequent queries.

2.3 Signature Generation

To generate a signature from a query spec, we cast it into a stably ordered representation and take a hash of those bytes.

As noted above, before hashing a query spec into its signature for a block, we remove the filters that are true for all records in the block. Filter elimination has the effect of normalizing the query to remove the time parameters if the whole block is contained within the query time window. (Figure 3)

To implement filter elimination, we iterate through all the integer filter predicates and remove all greater than and less than predicates that are true for the min and max values of that column in the block. This works because the greater than and less than operators are transitive.

2.4 Cache Policy

To avoid cache management headaches, the cache strategy is simple: query caching is enabled on a per query basis when requested, and the cache gets deleted on a regular basis.

We enable caching for queries that come from dashboards because we know dashboards contain repeated queries. Experimentally, we've also enabled caching for all queries that are issued through a UI and have seen good results.

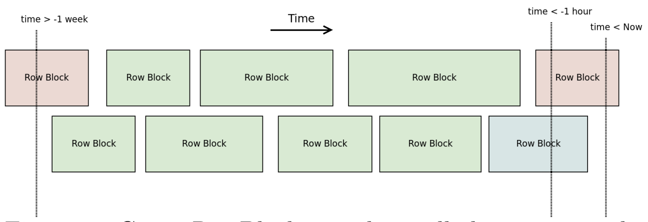


Figure 3: Green RowBlocks can have all their time predicates removed

3. PERFORMANCE

3.1 Methodology

To test performance, datasets were created in varying sizes: 100k, 200k, 500k, 1mm, 2mm, 5mm, 10mm, 20mm. The main columns generated were a time column (in seconds since the epoch), a ping column (a gaussian value) and a host column (chosen proportionally from a top500 domains listing).

We run each query in 3 configurations: a normal query without caching, a query that saves results to the cache and a query that re-uses cached results.

To prevent OS level file caching issues, we reset the file cache before running each of these queries using vmtouch.

To generate realistic data, timestamps are generated at about 1mm per day in mostly chronological order. The effect of this is that a given block will not have data that spans more than a few days of data, lowering the amount of results that a given block will contain for a GROUP BY BUCKET(time) query.

4. RELATED WORK

Techniques related to caching queries with timestamps predicates are to cache the query plan [3] for repeated use, caching intermediate results [4] and data aware caching [5]

However, modern database result caches have an important caveat: they do not support queries with dynamic functions, including date and timestamp functions [6] [7] [8] [9] and queries with slightly different timestamp predicates will not share cached results.

Our method of building a timestamp resistant query result cache combines two important ideas: 1) caching and re-using intermediate results instead of the final results and 2) rewriting the query specs on a per block basis to make them more cache friendly by eliminating filters.

The concept of filter elimination is strongly related to row-block elimination [2] - a method for only loading data relevant to the current time predicates.

5. CONCLUSIONS

Using an intermediate query result cache with filter elimination has an immediate benefit for repeated queries that involve timestamp filters.

Since query results are cached at the block level and reused during the aggregation phase during subsequent queries, the size of the result set will dictate the level of performance gains. For results where the cardinality of the per-block result set is small, there will be significant speed ups - therefore it makes sense to cache results based on a cardinality threshold of the result set.

Experimentally, the amount of time it takes to run and save the results of a query are rarely more than 2x the cost of the original query, while typical queries experience from 2 - 10x speed up.

6. REFERENCES

[1] Sybil - a fast and simple nosql olap engine. `https://github.com/logv/sybil`, 2018 (accessed August 20, 2018).

[2] A. Goel, B. Chopra, C. Gerea, D. MÃątÃąni, J. Metzler, F. Ul Haq, and J. Wiener. Fast database restarts at facebook. pages 541–549. ACM Press, 2014.

[3] B. Ramesh and M. Koppuravuri. Caching execution plans for queries that use timestamps, Aug 2012.

[4] P. Roy, K. Ramamritham, S. Seshadri, P. Shenoy, and S. Sudarshan. Don't trash your intermediate results, cache 'em. *arXiv:cs/0003005*, Mar 2000. arXiv: cs/0003005.

[5] Y. Zhao, J. Wu, and C. Liu. Dache: A data aware caching for big-data applications using the mapreduce framework. *Tsinghua Science and Technology*, 19(1):39âĂŞ50, Feb 2014.

[6] Database performance tuning guide. `https://docs.oracle.com/database/121/TGDBA/tune_result_cache.htm#TGDBA616`.

[7] Mysql 5.7 reference manualâĂŕ:: 8.10.3.1 how the query cache operates. `https://dev.mysql.com/doc/refman/5.7/en/query-cache-operation.html`.

[8] Using cached query results | bigquery. `https://cloud.google.com/bigquery/docs/cached-results`.

[9] Using persisted query results - snowflake documentation. `https://docs.snowflake.net/manuals/user-guide/querying-persisted-results.html`.

7. APPENDIX

7.0.1 5mm

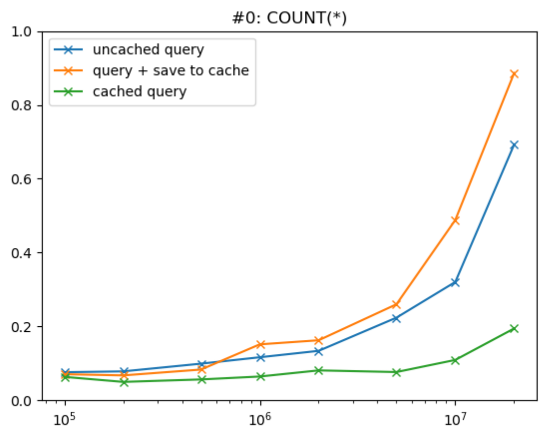
ID	Query	Cached Query	Uncached	Query + Save
0	COUNT(*)	0.0765	0.2237	0.2600
1	AVG(ping)	0.1593	0.8059	0.8082
2	HIST(ping)	0.1460	0.8140	0.8837
3	AVG(ping) GROUP BY BUCKET(time, 21600)	0.1137	0.9693	1.0931
4	GROUP BY host	0.2106	0.7682	0.9122
5	COUNT DISTINCT(host)	0.1074	0.8454	0.9801
6	COUNT DISTINCT(host) GROUP BY BUCKET(time, 21600)	0.2699	1.1067	1.6269
7	AVG(ping) GROUP BY host	0.3273	1.5142	1.9398
8	AVG(ping) GROUP BY host LIMIT 10	0.3147	1.4273	1.8983
9	HIST(ping) GROUP BY host	1.6130	2.9821	4.5891
10	AVG ping GROUP BY host WHERE host ˜= facebook|google	0.1498	1.3826	1.4467
11	GROUP BY host,status	0.3747	1.3424	2.6306
12	AVG ping GROUP BY host, status	0.9016	2.7560	4.3943
13	AVG ping GROUP BY host, BUCKET(time, 21600) LIMIT 10	0.9269	2.8435	4.5290
14	AVG ping GROUP BY host, BUCKET(time, 21600) LIMIT 100	1.4133	2.7796	4.6764

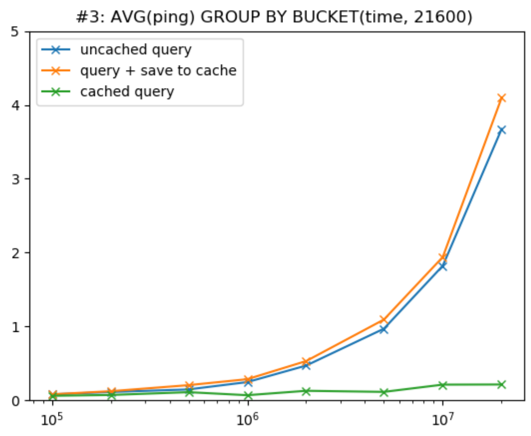
7.0.2 10mm

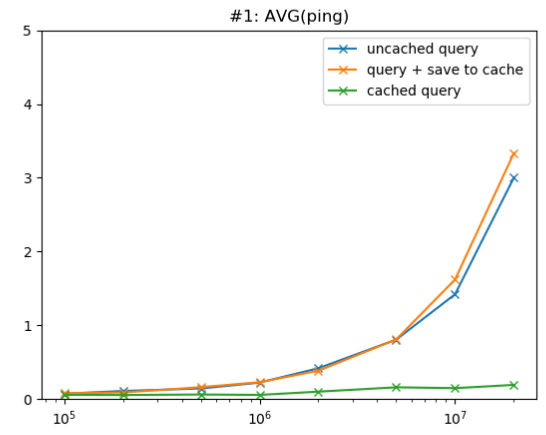
ID	Query	Cached Query	Uncached	Query + Save
0	COUNT(*)	0.1093	0.3204	0.4882
1	AVG(ping)	0.1480	1.4208	1.6207
2	HIST(ping)	0.1869	1.4995	1.6202
3	AVG(ping) GROUP BY BUCKET(time, 21600)	0.2119	1.8218	1.9402
4	GROUP BY host	0.3773	1.5690	1.9007
5	COUNT DISTINCT(host)	0.1510	1.6749	1.8963
6	COUNT DISTINCT(host) GROUP BY BUCKET(time, 21600)	0.4402	2.1909	3.1327
7	AVG(ping) GROUP BY host	0.5593	3.1028	3.7606
8	AVG(ping) GROUP BY host LIMIT 10	0.4805	2.9263	3.7481
9	HIST(ping) GROUP BY host	3.0954	6.0913	8.5119
10	AVG ping GROUP BY host WHERE host ˜= facebook|google	0.2431	2.7622	2.9934
11	GROUP BY host,status	0.7002	2.7353	5.1702
12	AVG ping GROUP BY host, status	1.7430	5.2002	8.8410
13	AVG ping GROUP BY host, BUCKET(time, 21600) LIMIT 10	1.7579	5.1599	9.0603
14	AVG ping GROUP BY host, BUCKET(time, 21600) LIMIT 100	2.5308	5.7232	10.2327

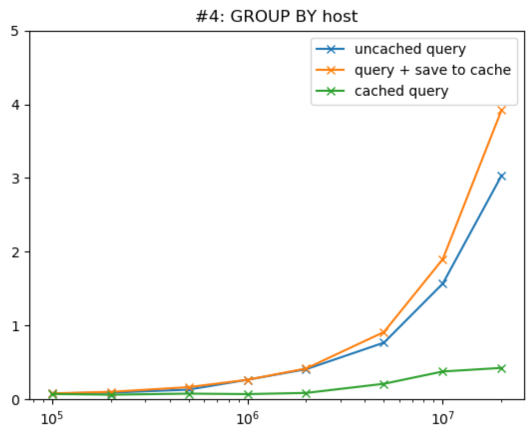
7.0.3 20mm

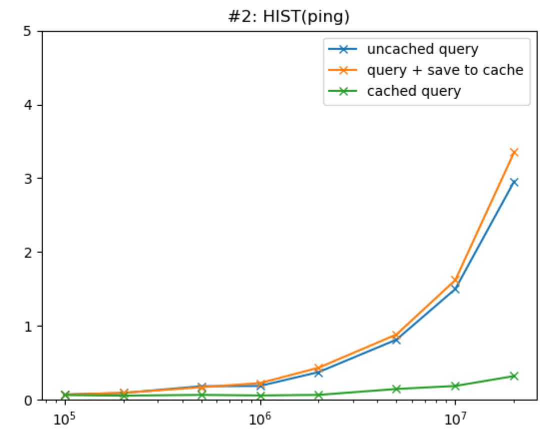
ID	Query	Cached Query	Uncached	Query + Save
0	COUNT(*)	0.1944	0.6928	0.8863
1	AVG(ping)	0.1922	3.0021	3.3357
2	HIST(ping)	0.3210	2.9564	3.3523
3	AVG(ping) GROUP BY BUCKET(time, 21600)	0.2137	3.6739	4.0977
4	GROUP BY host	0.4260	3.0361	3.9235
5	COUNT DISTINCT(host)	0.3922	3.1838	3.7552
6	COUNT DISTINCT(host) GROUP BY BUCKET(time, 21600)	0.5855	4.7212	6.2798
7	AVG(ping) GROUP BY host	0.9572	6.0674	7.6132
8	AVG(ping) GROUP BY host LIMIT 10	0.9121	5.9151	7.9821
9	HIST(ping) GROUP BY host	5.9496	11.3512	17.2638
10	AVG ping GROUP BY host WHERE host ˜= facebook|google	0.2952	5.5658	5.9463
11	GROUP BY host,status	1.1721	5.5759	10.3342
12	AVG ping GROUP BY host, status	3.4521	10.3238	18.5517
13	AVG ping GROUP BY host, BUCKET(time, 21600) LIMIT 10	3.3510	10.4989	19.0149
14	AVG ping GROUP BY host, BUCKET(time, 21600) LIMIT 100	5.5178	15.9722	23.9588

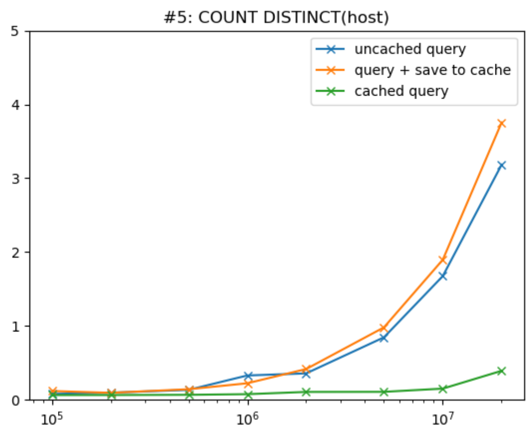

#0: COUNT(*)
uncached query
query + save to cache
cached query
1.0
0.8
0.6
0.4
0.2
0.0
10^5
10^6
10^7

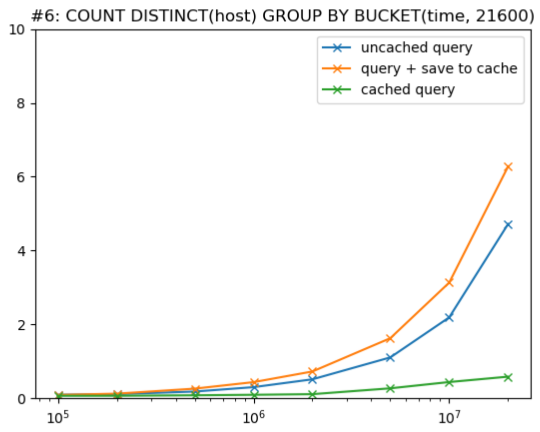

#3: AVG(ping) GROUP BY BUCKET(time, 21600)
uncached query
query + save to cache
cached query
5
4
3
2
1
0
10^5
10^6
10^7

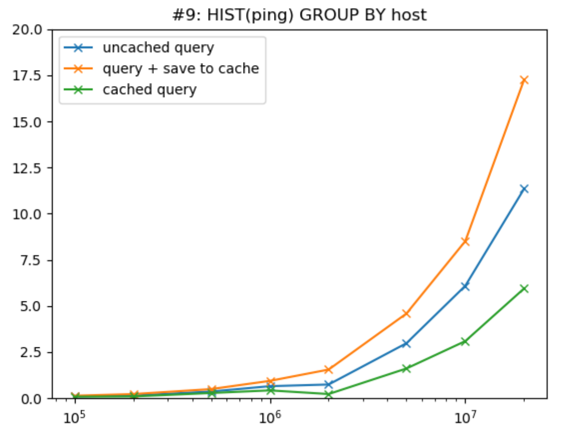

#1: AVG(ping)
uncached query
query + save to cache
cached query
5
4
3
2
1
0
10^5
10^6
10^7

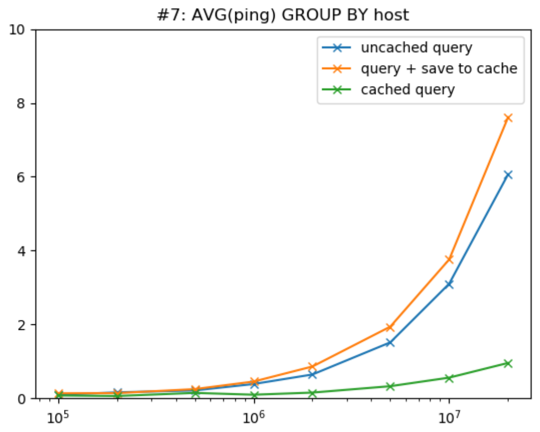

#4: GROUP BY host
uncached query
query + save to cache
cached query
5
4
3
2
1
0
10^5
10^6
10^7

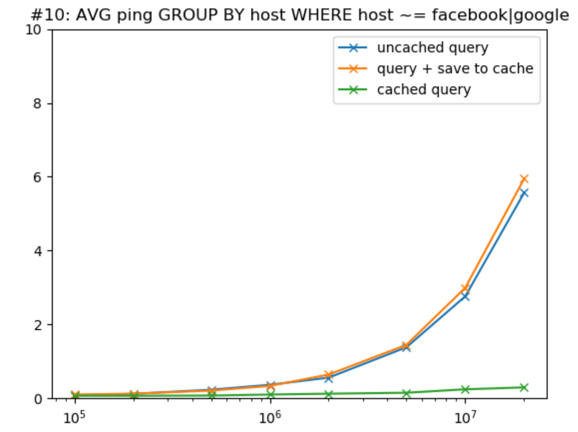

#2: HIST(ping)
uncached query
query + save to cache
cached query
5
4
3
2
1
0
10^5
10^6
10^7

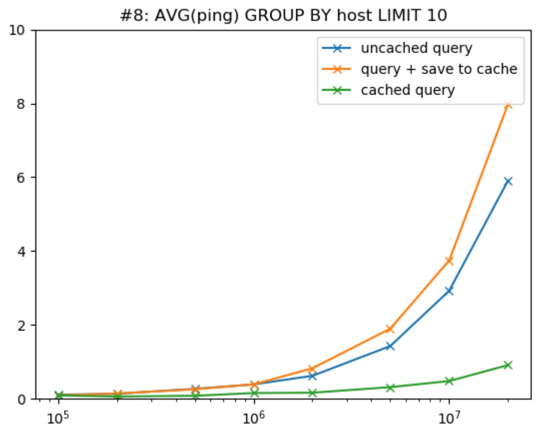

#5: COUNT DISTINCT(host)
uncached query
query + save to cache
cached query
5
4
3
2
1
0
10^5
10^6
10^7

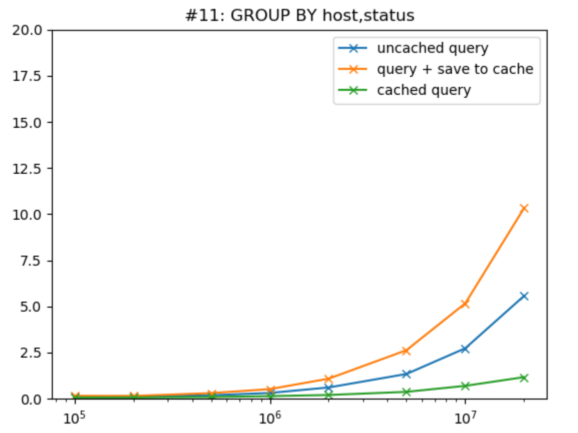

#6: COUNT DISTINCT(host) GROUP BY BUCKET(time, 21600)
uncached query
query + save to cache
cached query
0
2
4
6
8
10
10^5
10^6
10^7

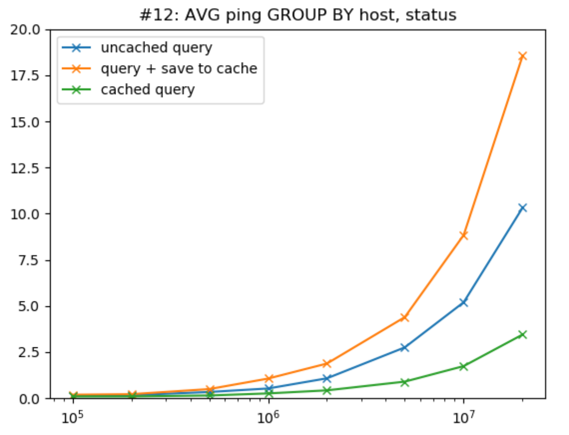

#9: HIST(ping) GROUP BY host
uncached query
query + save to cache
cached query
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
10^5
10^6
10^7

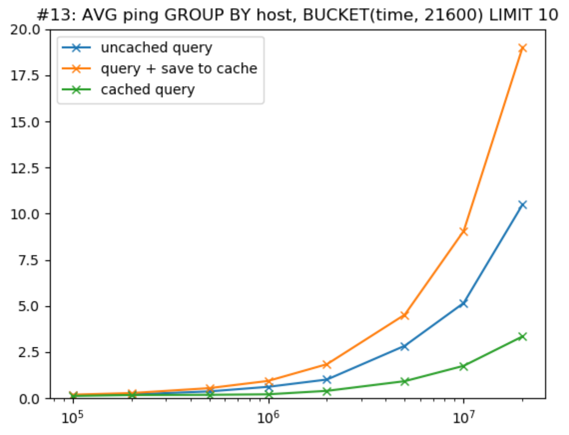

#7: AVG(ping) GROUP BY host
uncached query
query + save to cache
cached query
0
2
4
6
8
10
10^5
10^6
10^7

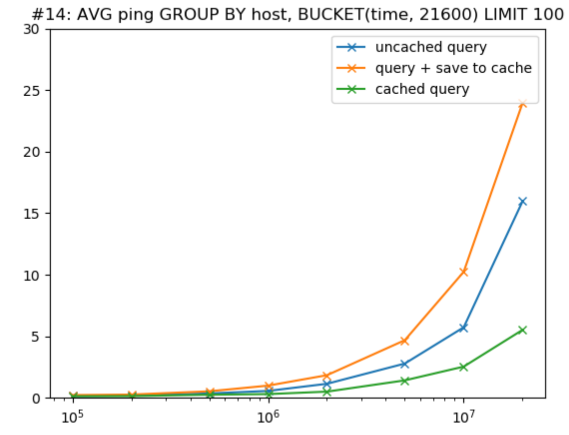

#10: AVG ping GROUP BY host WHERE host ~= facebook|google
uncached query
query + save to cache
cached query
0
2
4
6
8
10
10^5
10^6
10^7


#8: AVG(ping) GROUP BY host LIMIT 10
uncached query
query + save to cache
cached query
0
2
4
6
8
10
10^5
10^6
10^7


#11: GROUP BY host,status
uncached query
query + save to cache
cached query
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
10^5
10^6
10^7


#12: AVG ping GROUP BY host, status
uncached query
query + save to cache
cached query
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
10^5
10^6
10^7


#13: AVG ping GROUP BY host, BUCKET(time, 21600) LIMIT 10
uncached query
query + save to cache
cached query
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
10^5
10^6
10^7


#14: AVG ping GROUP BY host, BUCKET(time, 21600) LIMIT 100
uncached query
query + save to cache
cached query
30
25
20
15
10
5
0
10^5
10^6
10^7