Scalable User Sampling on audience data

Guruganesh Kotta, Michael Natkovich, Nathan Speidel

How user sampling is currently performed at Yahoo

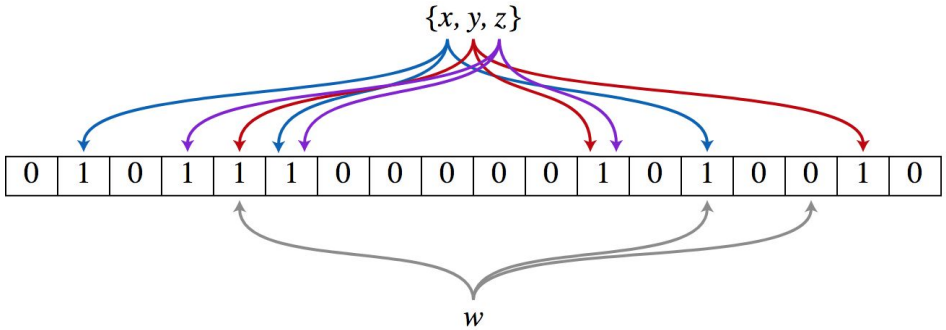
- Data analysts often need to track the behavior of a cohort of users
- Querying all of user data is often not feasible when tracking a cohort users over a long period of time for several reasons
  - poor query performance
  - high resource consumption
- Currently, analysts have 2 options
  - Create their own sampled feed with a subset of data
    - Creates an unnecessary amount of work and overhead for our analysts
  - Run their queries on all audience data
    - Have to deal with poor performance
- We wanted to provide a solution that won't require the creation of additional data sets, while providing a reasonable performance improvement compared to a full table scan.

Sampling approach

- We worked with audience datasets, which includes any user interaction data, and no advertising data
- In this dataset, a string uuid serves as a unique identifier for a user
- Created a sampling column based on a hash of the unique user id
- 4 different bucket sizes
  - 0.1% sample (1000 buckets), 0.01% sample (10k buckets), 0.001% sample (100k buckets), and a 0.0001% sample (million buckets)
  - Can also specify a range of buckets for other sample sizes.
    - Selecting buckets 1-10 would give a 1% sample
- After adding the sampling column, we wanted to explore ways to optimize queries to increase performance
- We came up with 2 approaches:
  - Using bloom filters
  - Sorting the data in each file by the sampling column

Bloom filters

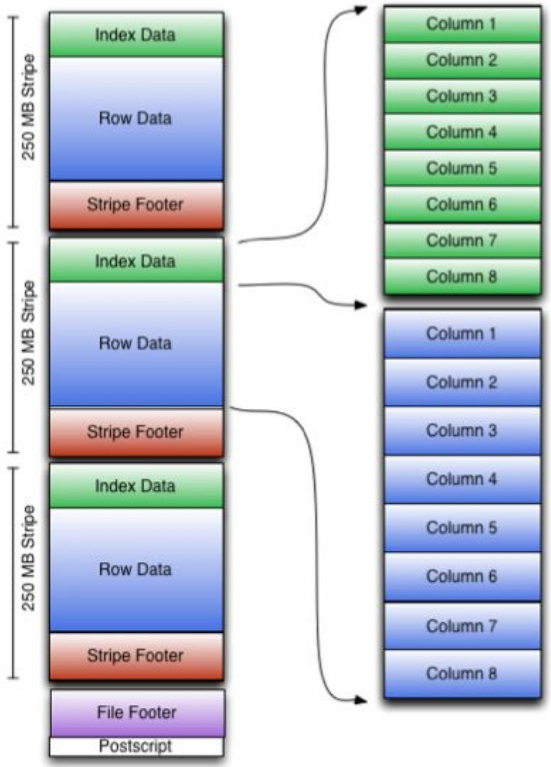
- Used to quickly filter out large amounts of users
- The bloom would figure out a bucket was not part of our query, and if so, it would be able to easily filter out any data associated with that bucket.
- For our use case, we would not run into false positives
- We theorized that this method will be most effective when using a small sample size



YAHOO!

Sorting

- We tried sorting our data files by the sampling column.
- Query performance can be optimized if we can skip over files which do not contain data relevant to the query.
- The entry with the lowest bucket id is stored at the beginning of the file, and the entry with the largest bucket id is stored at the end of the file.
- By ordering each file in this way, the query can be optimized to easily skip over large parts of the file.
- Each file can store some index metadata which contains the minimum and maximum bucket id in that file.

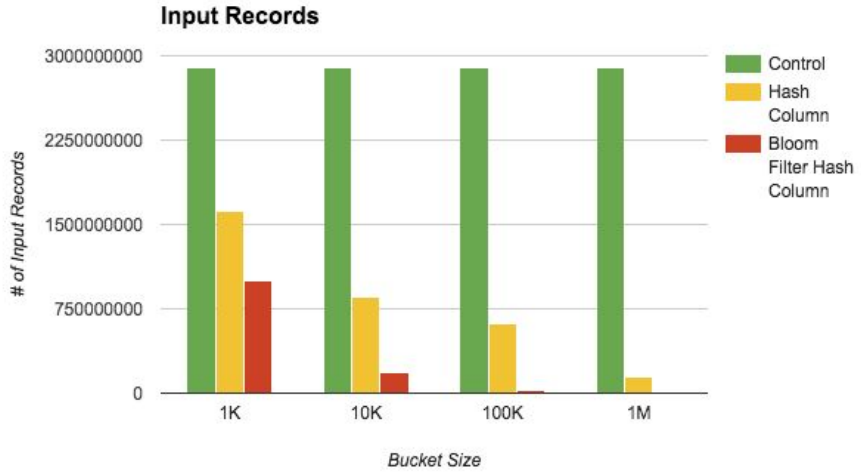


YAHOO!

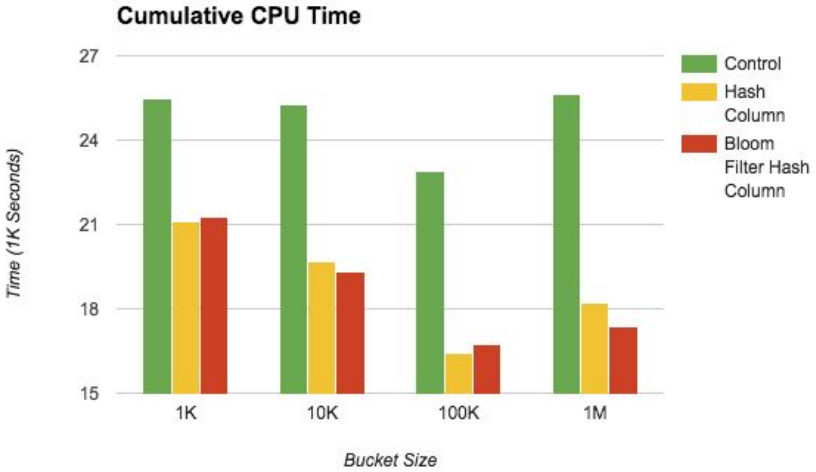
Testing methodology

- We tested these methods using several different tables
  - 1 control table
  - 1 database with a sampling column, but no bloom filters or sorting
  - 1 database with blooms enabled on the sampling column
  - 1 database with sorting enabled on the sampling column
  - Each database included 4 tables, each with a different number of buckets
    - 1k buckets, 10k buckets, 100k buckets, 1m buckets
- We recorded the following metrics
  - CPU Time
  - Input records

Bloom Filter Results



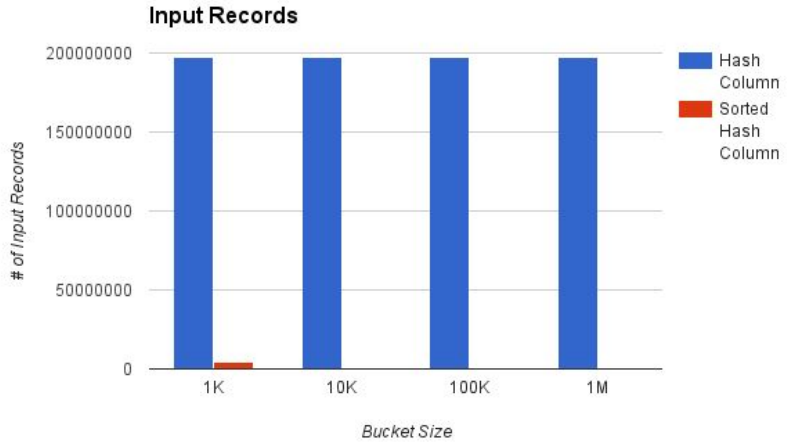
99.9% decrease



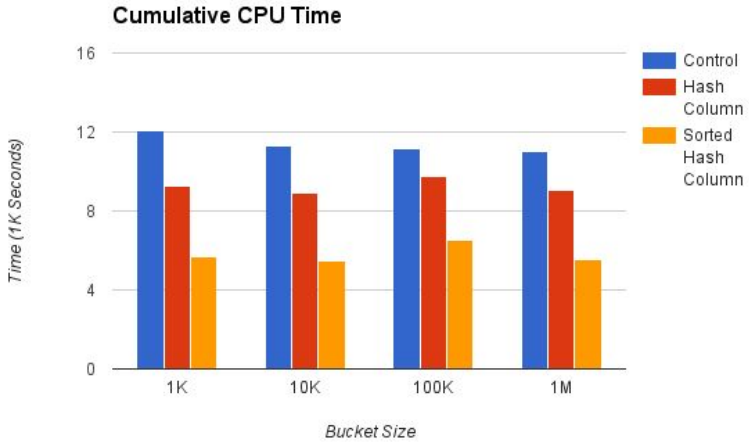
15-35% speedup

YAHOO!

Sorting Results



99.9% decrease



50-60% speedup

YAHOO!

Conclusions

- Using sorting resulted in significant performance improvements
- Using bloom filters resulted in some improvement
- Sorting provided a consistent performance improvement across all bucket sizes
- Bloom filters performed better with large bucket sizes, such as one million buckets
- For our use case, sorting is the better solution
  - It resulted in a good performance speedup for all bucket sizes
  - We think that our users would like to use smaller bucket sizes in particular