Is Partitioning Really Worth It?
Efficient, Fully Concurrent Hash Aggregation

Daniel Xue, Ryan Marcus | DB@Penn

University of Pennsylvania

Motivation

1. Server-grade hardware has been steadily increasing in core count, making intra-query parallelism crucial for performance gains.
2. Hash aggregations have proven tricky to parallelize owing to the traditionally poor performance of concurrent hash tables. Therefore, many systems have taken a partitioned approach [1].
3. Partitioning-based systems can suffer from scalability issues due to data skew and merging overhead. Can a fully concurrent approach overcome these challenges?

Ticketing

Ticketing uses a hash table and a (fuzzy) counter shared between all threads. Note that once a key has been inserted, it is never deleted, and its ticket value is never updated. While general-purpose concurrent hash tables can have poor performance, we find that hash tables optimized around insert and lookup only workloads achieve excellent throughput and scalability. We benchmark various hash table designs including an insert and lookup only variation of folkloreHT [2], hopscotch hashing [3], cuckoo hashing [4], a linear probing hash table where each entry is a one-time lock, as well as a globally locked hash table as a baseline.

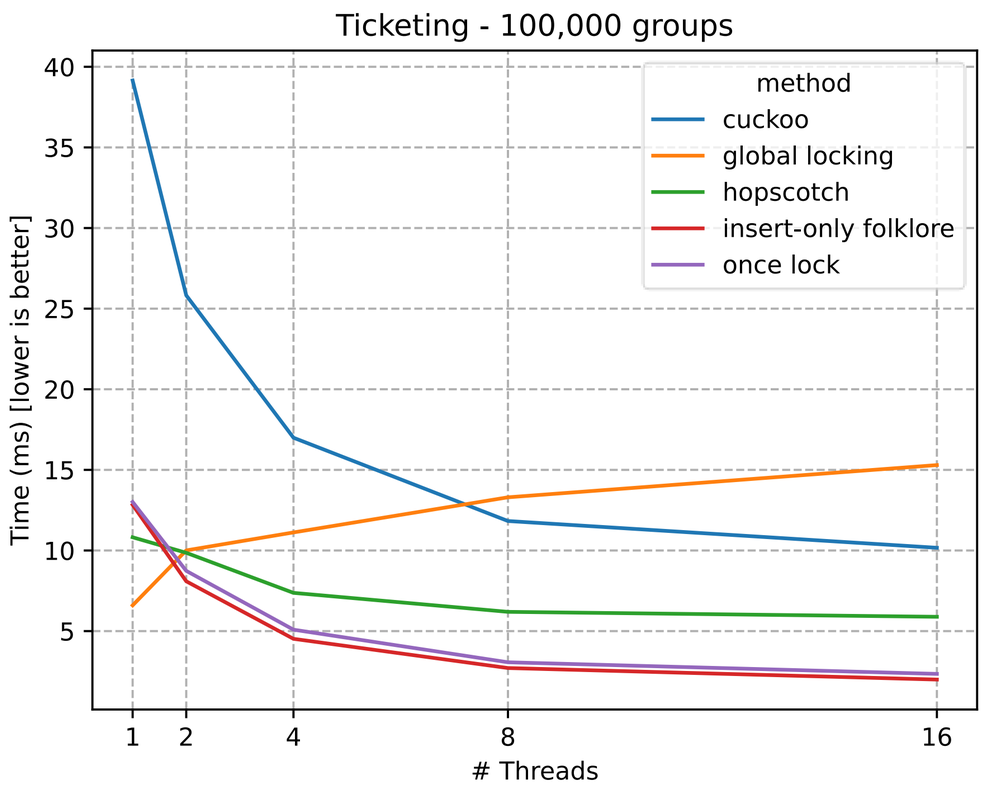


Figure 2: ticketing performance and thread scaling using various hash table implementations. 1 millions rows are ticketed.

Future Work

1. Test hash table resizing strategies that avoid blocking, such as multi-threaded resizing, cached lookup-only tables while resizing, and/or a temporary overflow table.
2. Implement within an existing DBMS that uses partitioning for a more realistic head-to-head comparison.
3. Benchmark using real-world workloads (e.g. JOB).

github.com/danielxue/nedb25-poster

References

[1] V. Leis, P. Boncz, A. Kemper, and T. Neumann, “Morsel-driven Parallelism: a NUMA-aware Query Evaluation Framework for the Many-core Age,” SIGMOD 14.
[2] T. Maier, P. Sanders, and R. Dementiev, “Concurrent Hash Tables: Fast and General(?)!,” TOPC 19.
[3] R. Kelly, B. A. Pearlmutter, and P. Maguire, “Lock-Free Hopscotch Hashing,” APOCS 20.
[4] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, “Algorithmic Improvements for Fast Concurrent Cuckoo Hashing,” EuroSys 14.

Approach

We split the aggregation process into two stages.
1. Ticketing: map every unique key into a unique (roughly incrementing) integer (its ticket).
2. Aggregation: use the tickets to index into an array of cumulative aggregation values, updating as needed.

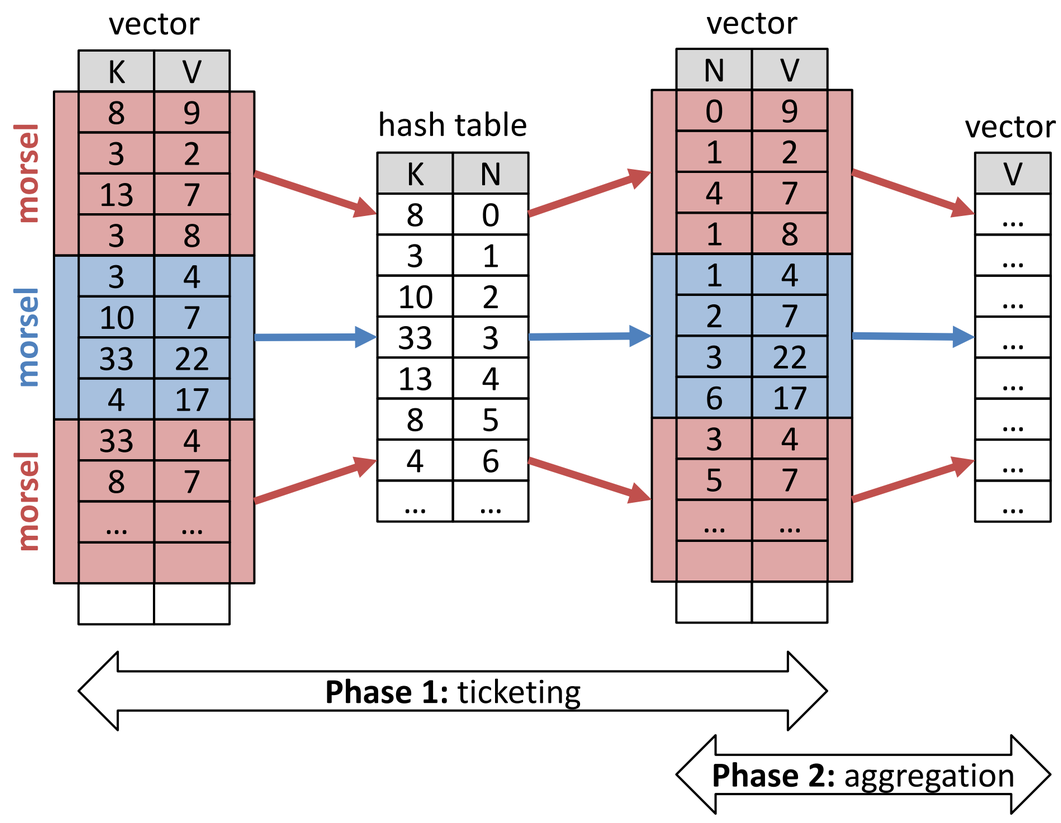
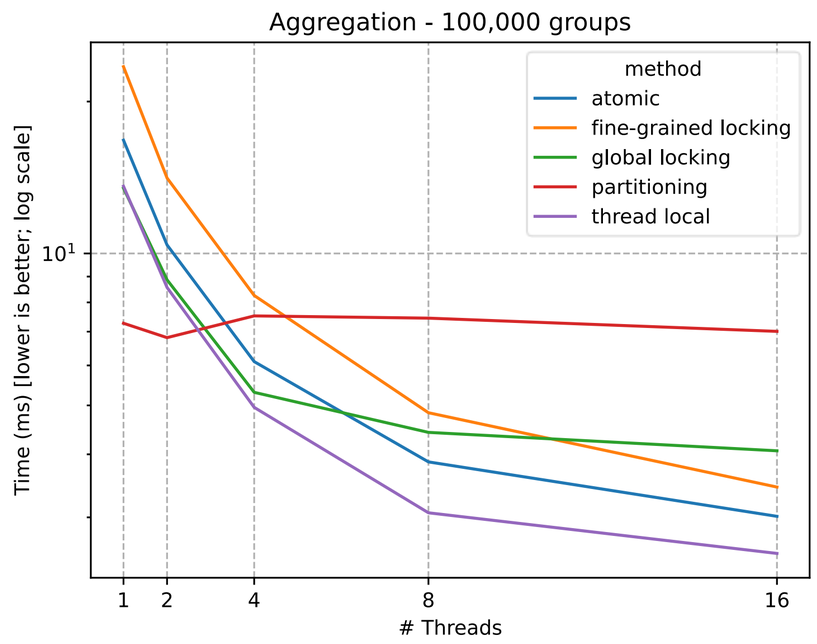


Figure 1: our fully concurrent two-step hash aggregation, using the same instance as the partitioned method in [1] to demonstrate the difference in approach.

Aggregation

To aggregate, we maintain a vector of cumulative aggregated values indexed by ticket. The full process is benchmarked with various concurrency management strategies (and uses our variation of folkloreHT for the ticketing step).



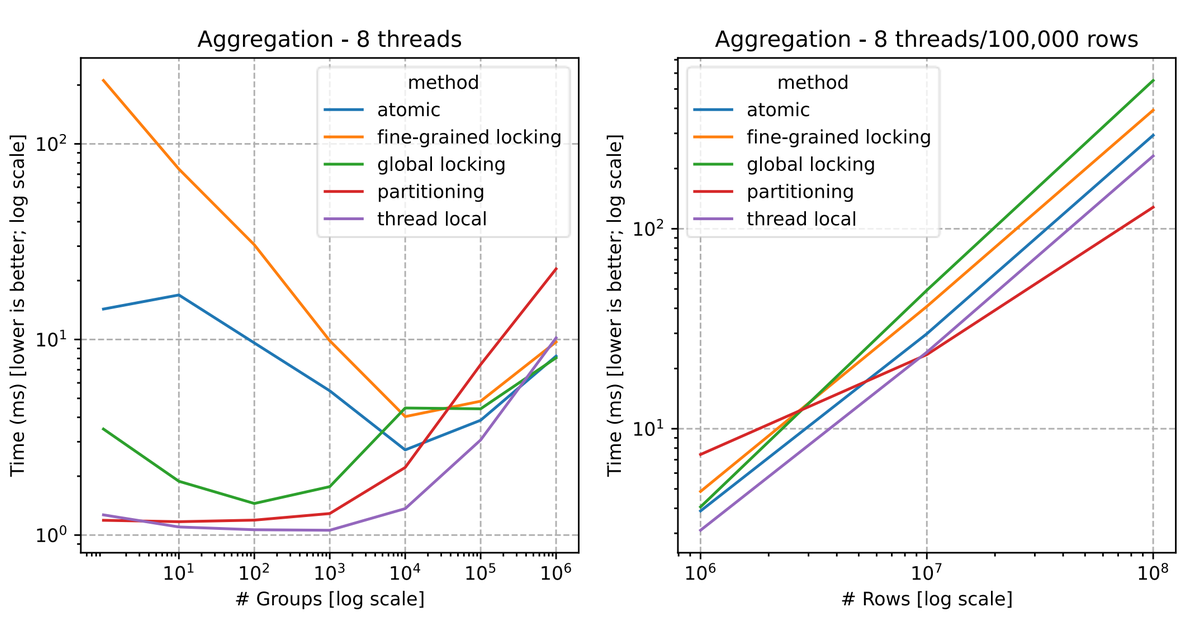


Figure 3: aggregation performance with different concurrency management methods and varied thread, group, and row count. Unless otherwise indicated, 1 million rows are aggregated. Note time is on a log scale, as are rows and groups.