Efficient Sorting, Duplicate Removal, Grouping, and Aggregation

Thanh Do, Goetz Graefe, and Jeffrey Naughton (2022)

Adam Janicki
April 25th, 2023

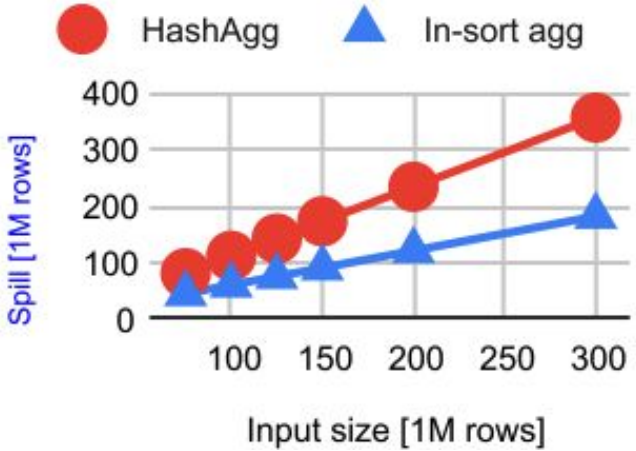
A better title

“Improving the performance of duplicate removal, grouping, and aggregation queries via efficient sorting.”

Contributions

- Developed 2 new techniques to improve *sorting-based* aggregation, grouping, and duplicate deletion queries: **early aggregation** and **wide merging**.
- Sorting-based aggregation is made competitive with hashing-based aggregation.
- Eliminates the need for query engine to pick an algorithm before execution by having one algorithm that is always optimal.

Comparison of spillage (amount of external memory used) by hashing, and early aggregation (in-sort) techniques



Motivation

- Removing duplicates, grouping, and aggregations are very common relational database operations.
- Authors come from perspective of Google’s F1 Query (SQL at massive scale), so efficiency matters significantly for large data.
- Currently, there exist a few different algorithms to execute these queries, but they are optimal under different circumstances.

Talk Overview

1. Background
2. Technique 1: Early Aggregation
3. Technique 2: Wide Merging
4. Performance Results
5. Conclusion

Removing Duplicates

- Get rid of duplicates based on a condition.
- SELECT DISTINCT name FROM users;
- This query returns all unique names from the table of users.

Username	Name
ajanicki	Adam
rose4	Rose
adam2	Adam
mike_1	Mike

Name
Adam
Rose
Mike

Grouping

- Organize the output by a set of columns.
- SELECT name, username FROM users GROUP BY name;
- This query returns all usernames and names which are organized into contiguous chunks of entries with the same name.

Username	Name
ajanicki	Adam
rose4	Rose
adam2	Adam
mike_1	Mike

Username	Name
ajanicki	Adam
adam2	Adam
rose4	Rose
mike_1	Mike

Aggregation

- Calculates additional data based on the table.
- Examples include sum, count, avg, etc.
- SELECT name, COUNT(*) AS frequency FROM users GROUP BY name;

Username	Name
ajanicki	Adam
rose4	Rose
adam2	Adam
mike_1	Mike

Name	frequency
Adam	2
Rose	1
Mike	1

Current methods

- Two primary ways: hashing-based, and sorting-based.
- Each come with benefits and drawbacks, and work better in different scenarios.
- Currently, one of them is picked before query execution based on estimation of which one is better.

Hashing

- A divide and conquer algorithm that hashes rows into disjoint subgroups.
- Allows for removal of duplicates on the fly, reducing memory usage.
- Hashing is the optimal choice if the in-memory size M is greater than the output size O

Traditional Sorting

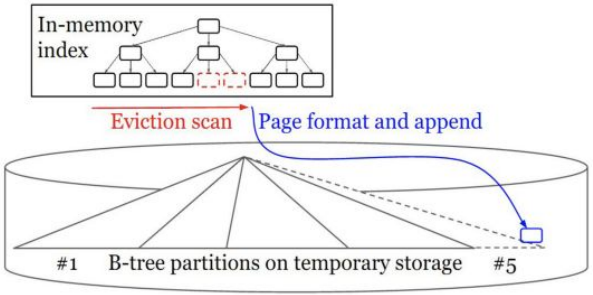
- Also a divide and conquer algorithm, this approach works by applying a merge sort to the elements.
- Sorting can produce *interesting orderings*, which are orderings produced by sorting which speed up subsequent join and grouping operations.
- Sorting is the optimal choice if the in-memory size M is less than the output size O, and if interesting orderings would benefit the following queries.

Efficiency Factors

- When evaluating the methods for querying, there are two main factors of efficiency that are considered.
- Execution/CPU time, which is the speed that queries can be executed.
- Spillage (amount of external memory used), which matters a lot more for large databases where space becomes an issue.

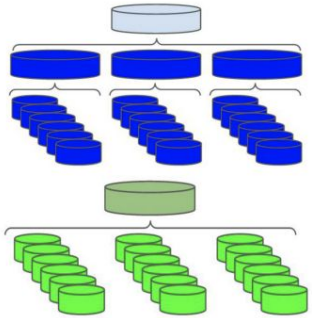
Technique 1: Early Aggregation

- Early aggregation feeds into the input of the external merge sort, reducing the amount of work the sort has to do.
- Uses a data structure, such as a B-tree, to create and maintain a sorted index of rows.
- Similar to a hash table, duplicate values can be detected and absorbed on insertion into this index.
- In the case where the output cannot fit in memory (O > M), scans will evict items from the B-tree into temporary external memory.



Technique 2: Wide Merge

- Wide merge takes effect during the merge step of the external merge sort.
- It means the fan-in (input) on the last merge in the sort is not specified, which eliminates memory used for intermediate merge steps.
- Instead of using a separate page for each buffer for each input, it uses one single buffer, spanning all available memory, to store inputs to the merge.
- Only applies to the final merge step, sometimes traditional merges are needed for intermediate steps.
- Reduces the amount of temporary memory needed by the sort by eliminating waste.

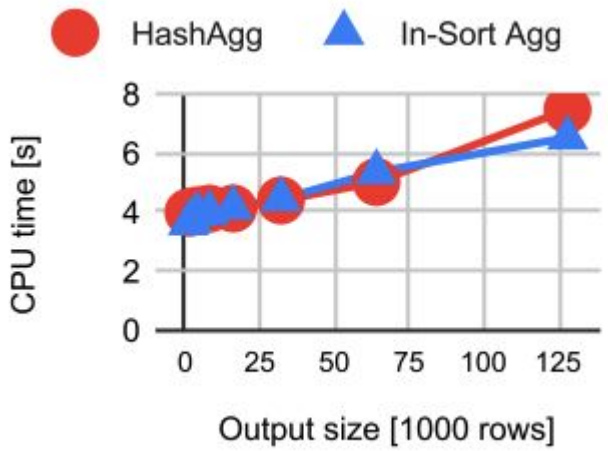


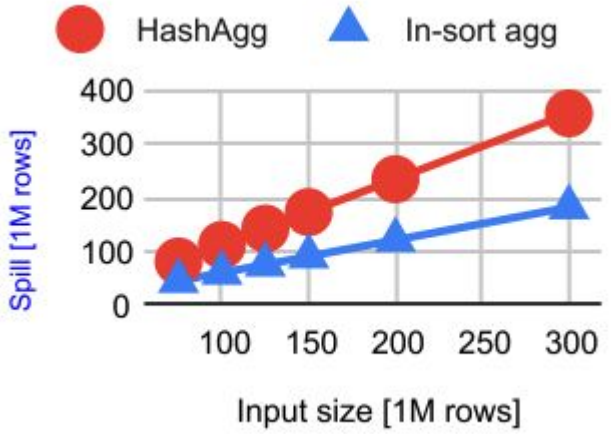
Authors’ Hypothesis

- Traditional Sorting is not competitive with hashing because it cannot use early aggregation like hashing can, so it doesn’t work well when M > O.
- The new approach *is* competitive with hashing by using early aggregation and wide merging.
- The new sorting approach is superior to hashing if interesting orderings, produced by sorting but not hashing, matter for making future operations faster.

Early Aggregation Performance

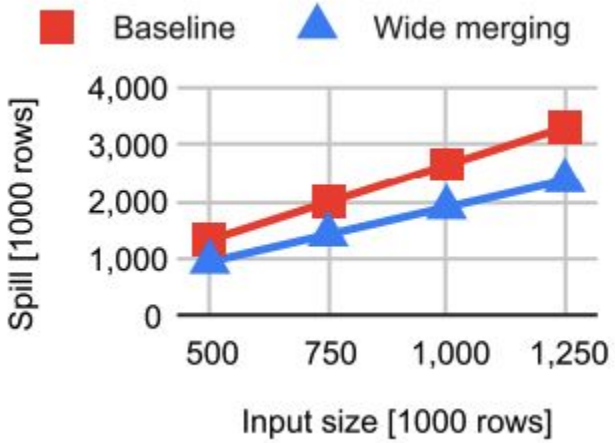
- Figures show a comparison of spillage (amount of external temporary memory used) at different input sizes





Wide Merge Performance

- Figure shows a comparison of spillage



Benefits of the new sorting approach

- Now sorting is just as efficient as hashing.
- It has the additional benefit of producing interesting orderings for the sort order.
- Only one algorithm is needed, which eliminates the engine having to predict which algorithm is more efficient before execution.
- Significantly reduces code size, which also reduces engineering work, and chances that bugs are introduced.
- Improves modularity and portability overall — good software engineering as well as performance engineering!

Conclusion

- Important to recognize that simpler solutions provide benefits beyond performance engineering.
- How could parallelization be applied to improve performance?
- Paper is new, so unclear where it will be applied beyond Google.
- Will other places attempt to adopt this novel approach to making these operations faster?
- Are there any other systems where efficiency can be gained simply by reducing complexity?