Efficient Sorting, Duplicate Removal, Grouping, and Aggregation

Presented By Louise He

Background

Senior Staff Software Engineer
Tech Lead Manager
@ Google F1

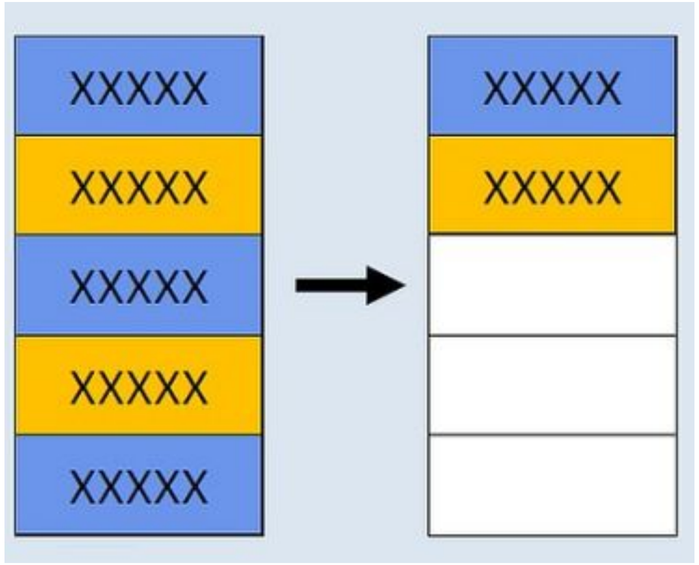
Principal Scientist
@ Google

Computer Scientist,
Former Professor and
Department Chair of
Computer Sciences
@ University of
Wisconsin–Madison,

Problem Trying to Solve

Duplicate Removal

- Eliminating repeated entries from a dataset
- The operation ensures that each unique value is retained only once
- It is commonly used in database queries such as SELECT DISTINCT or COUNT(DISTINCT ...)



Problem Trying to Solve

Grouping

- Grouping divides data into subsets based on one or more attributes
- This allows each group to be processed or analyzed separately, such as through aggregation

Animal	Habitat
Lion	Savanna
Zebra	Savanna
Penguin	Polar
Seal	Polar

Habitat	Count
Savanna	2
Polar	2

Problem Trying to Solve

Aggregation

- Performing summary computations on grouped data in a database query
- Commonly involves applying aggregation functions such as COUNT, SUM, AVG, MIN, and MAX
- Allows extraction of meaningful statistics from each group after the grouping operation

ID	Species
a001	Cat
a002	Dog
a003	Cat
a004	Bird

$\Rightarrow$

Species	Frequency
Cat	2
Dog	1
Bird	1

Current Methods

Hash-based Aggregation

- **Advantage:** Widely regarded as supporting early aggregation, meaning that identical keys can be combined immediately while building the in-memory hash table, reducing intermediate data volume.

- **Disadvantages:**
  - If the output exceeds available memory, the entire partition must spill to disk, leading to significant I/O overhead
  - The output is not ordered; if a subsequent operator requires sorted input (e.g., merge join), an additional sorting step is needed, increasing total cost

Current Methods

Sort-based Aggregation

- **Typical workflow: “external merge sort + in-stream aggregation”**
  - Perform multiple rounds of external sorting to generate sorted runs
  - Merge these runs across multiple stages to produce a globally ordered output
  - Apply deduplication or aggregation during the last merge phase

- **Disadvantages:**
  - No effective aggregation can occur before sorting completes, causing large amounts of unaggregated data to be written to temporary storage and participate in I/O
  - Overall I/O cost is high, and performance degrades significantly when the output is large
  - Widely considered that “sorting cannot achieve early aggregation,” making it less efficient than hash-based methods

Current Methods - Problems

- **Dependence on accurate cardinality estimates:**
  Rely heavily on estimating the output size accurately. When the estimate is wrong, the optimizer may choose an inappropriate algorithm, causing performance issues

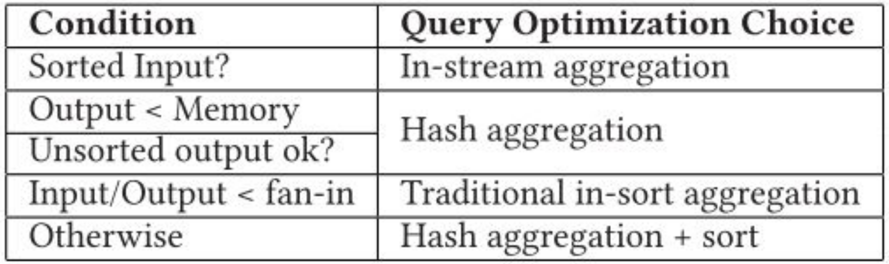
- **Insufficient memory utilization:**
  Inaccurate estimates often lead to mistakenly choosing an external algorithm, resulting in unnecessary I/O overhead

- **Lack of robustness:**
  In large-scale data scenarios, they may require recursive partitioning or multi-level merging, increasing both complexity and cost

- **Ignoring the impact of modern hierarchical storage:**
  Current analyses overlook the effects of multi-tier storage systems and fail to exploit pre-aggregation opportunities in parallel execution, limiting overall optimization potential

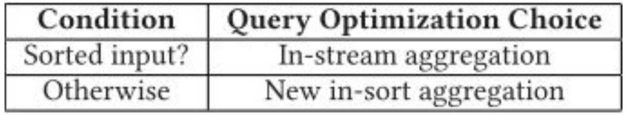
Current Methods

Table 1. Traditional Decision Procedure



Condition	Query Optimization Choice
Sorted Input?	In-stream aggregation
Output < Memory	Hash aggregation
Unsorted output ok?	
Input/Output < fan-in	Traditional in-sort aggregation
Otherwise	Hash aggregation + sort

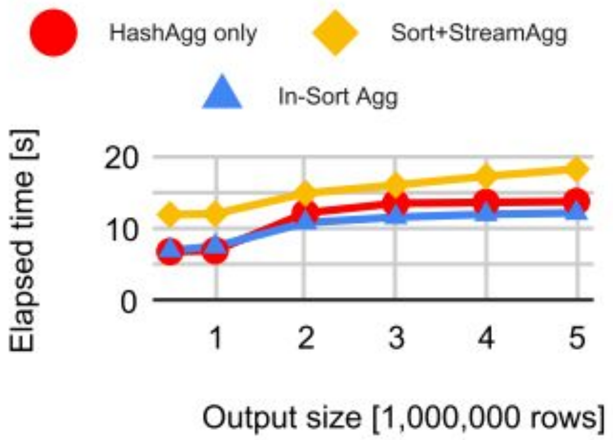
Table 2. Decision Procedure with the New Algorithm



Condition	Query Optimization Choice
Sorted input?	In-stream aggregation
Otherwise	New in-sort aggregation

Proposed Method

- Develop a new **in-sort** aggregation algorithm that can replace both traditional sort-based and hash-based aggregation
- Ensure the new algorithm delivers performance competitive with hash aggregation, even for large unsorted inputs
- Produce sorted output as part of the aggregation process, enabling downstream operators to benefit from the ordering



Early Aggregation

- Early Aggregation performs “merge-as-you-read” using an in-memory index at the start of sorting, preventing large amounts of duplicate data from being written to disk and significantly improving the efficiency of deduplication, grouping, and aggregation

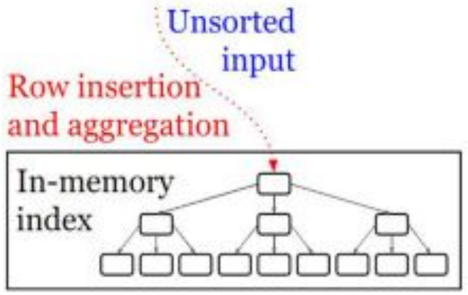


Fig. 5. In-memory aggregation.

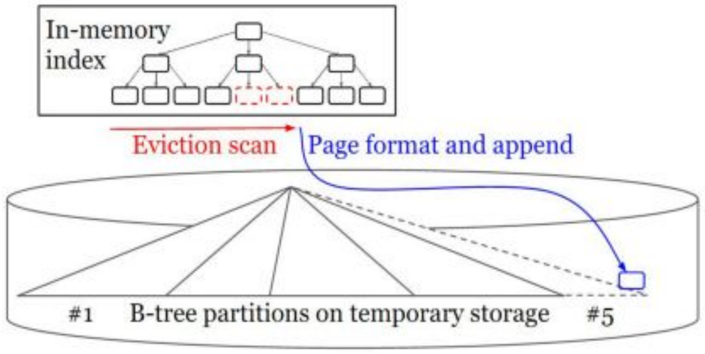
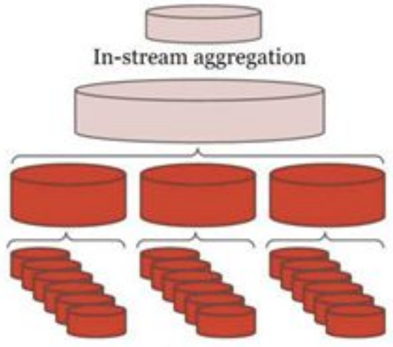
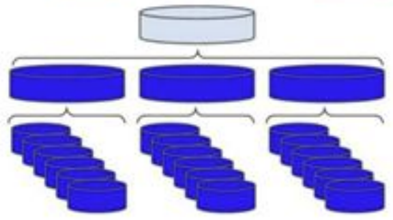


Fig. 6. Run generation using an ordered in-memory index.

Early Aggregation

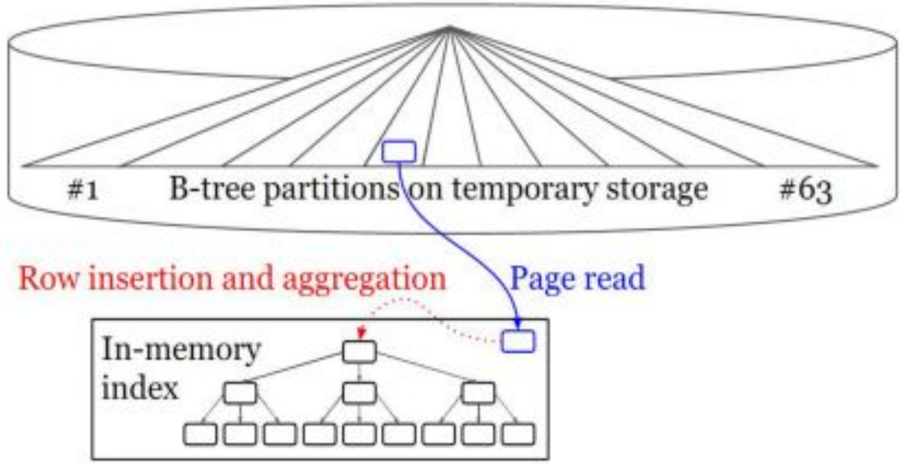
- Reduces disk I/O
- Aggregate-as-you-read
- Ideal for small-output workloads
- Shrinks temporary data volume
- Comparable I/O to hash aggregation
- Supports flexible implementations
- Unifies core data structures





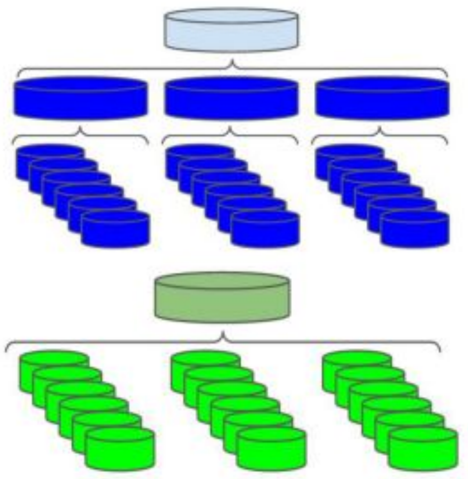
Wide Merging

- By using a single in-memory index and one input buffer, the algorithm merges a large number of runs in a single final merge step, avoiding multi-level intermediate merges and thereby reducing I/O overhead while achieving performance comparable to, or better than, hash aggregation


#1
B-tree partitions on temporary storage
#63
Row insertion and aggregation
Page read
In-memory index

Wide Merging

- Can process a large number of runs in a single merge step with a single input buffer together with an in-memory index
- Eliminates multi-level intermediate merges and significantly reduces overall I/O overhead
- Total amount of spilled data is lower than in traditional approaches



Performance - Early Aggregation

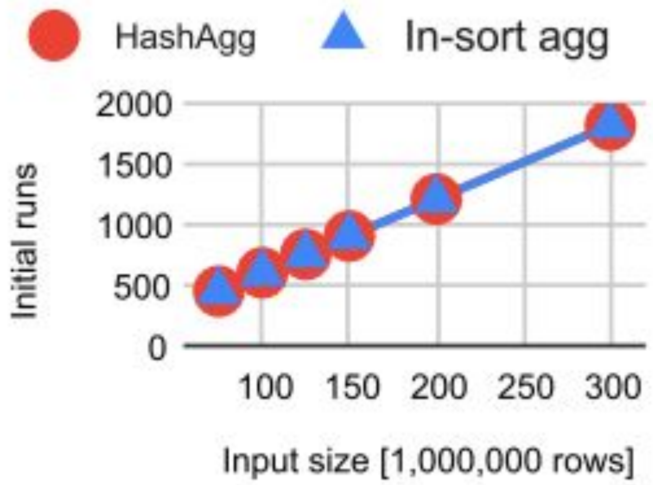


Fig. 17. Count of runs spilled from memory to temporary storage.

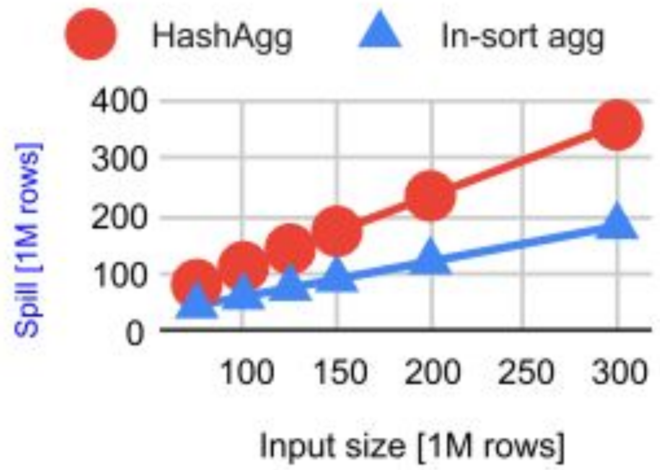


Fig. 18. Spill volume.

Performance - Wide Merging

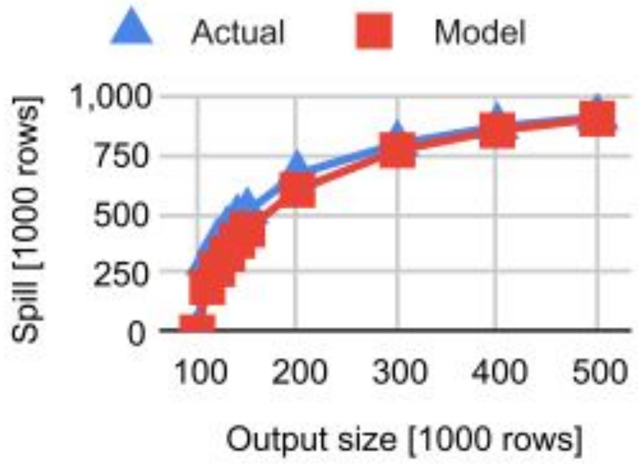


Fig. 12. Spill volume to runs on temp. storage.

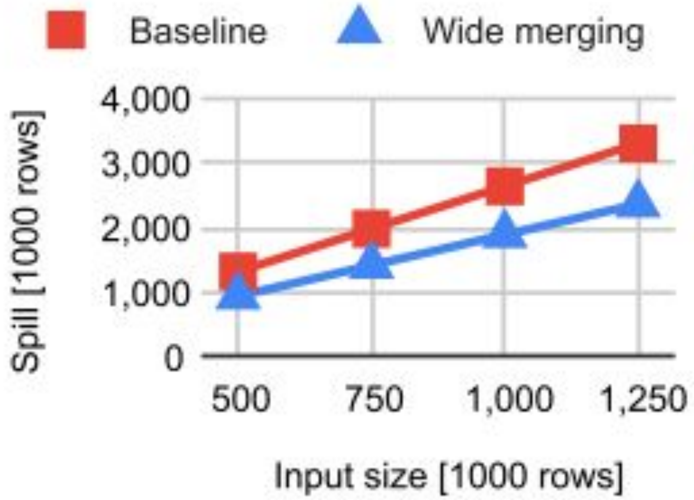


Fig. 14. Effect of wide merging.

Performance - General (CPU)

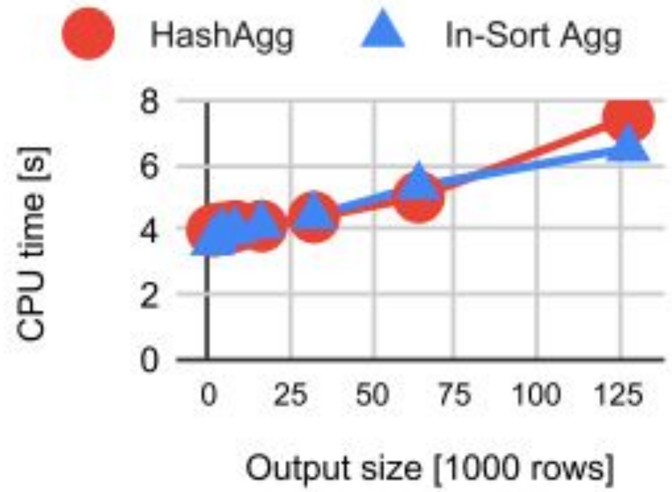


Fig. 15. Performance of in-memory indexes.

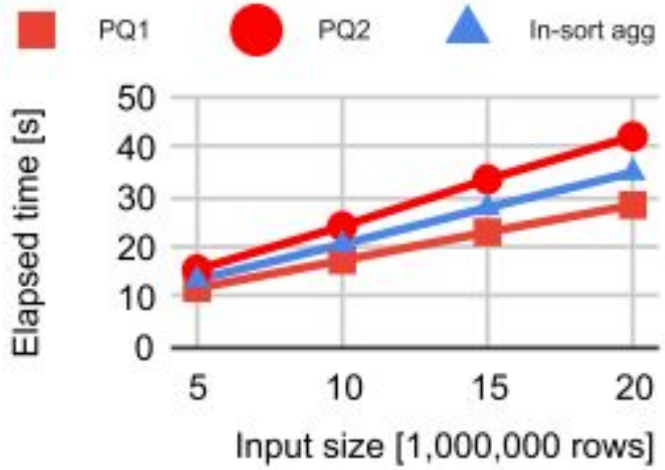


Fig. 16. Performance of run generation. "PQ1" and "PQ2" use tree-of-losers priority queues optimized with offset-value coding and normalized keys, respectively.

Strengths

- Early aggregation significantly **reduces disk I/O**, especially when the output is much smaller than the input
- The number of initial runs is comparable to hash aggregation, **avoiding additional spills**
- Wide merging minimizes multi-level merges and **further lowers I/O cost**
- CPU overhead is similar to hash aggregation, adding no meaningful computation burden
- The method produces naturally sorted output, accelerating downstream operators like merge joins and window functions
- **Spill behavior is highly predictable**, making plan selection easier for the optimizer
- The approach is more **robust** under data skew and less sensitive to cardinality estimation errors

Weaknesses

- **Harder to implement**
  It is more complicated to build and maintain than hash aggregation
- **Less helpful when output ≈ input**
  If the output size is almost as large as the input, early aggregation cannot reduce much data
- **B-tree maintenance costs more than a hash table**
  Keeping an ordered B-tree in memory involves more overhead than maintaining a hash table
- **No clear advantage on small datasets**
  When the data volume is very small, the method performs similarly to hashing and does not provide obvious benefits
- **Wide merging needs enough memory**
  If memory is too limited, wide merging cannot work effectively, and its performance advantage decreases.

Potential Next Step? / Discussions

- Can we make in-sort aggregation more adaptive, for example by detecting data skew or changes in output size on the fly and adjusting the index structure, spill policy, or merge strategy accordingly?

- Could there be a hybrid aggregation strategy that dynamically switches between hash-based and in-sort aggregation based on runtime statistics (e.g., skew, cardinality, memory pressure)?

- Since the current method relies on an in-memory ordered index, could alternative structures such as tries, skip lists, learned indexes, or ART improve insertion cost or cache locality?