Sort-based grouping and aggregation

In-memory b-trees for run generation and merging

Thanh Do
Google Inc
Madison, WI
tddo@google.com

Goetz Graefe
Google Inc
Madison, WI
goetzg@google.com

ABSTRACT

Database query processing requires algorithms for duplicate removal, grouping, and aggregation. Three algorithms exist: in-stream aggregation is most efficient by far but requires sorted input; sort-based aggregation relies on external merge sort; and hash aggregation relies on an in-memory hash table plus hash partitioning to temporary storage. Cost-based query optimization chooses which algorithm to use based on several factors including input and output sizes, the sort order of the input, and the need for sorted output. For example, hash-based aggregation is ideal for small output (e.g., TPC-H Query 1), whereas sorting the entire input and aggregating after sorting are preferable when both aggregation input and output are large and the output needs to be sorted for a subsequent operation such as a merge join.

Unfortunately, the size information required for a sound choice is often inaccurate or unavailable during query optimization, leading to sub-optimal algorithm choices. To address this challenge, this paper introduces a new algorithm for sort-based duplicate removal, grouping, and aggregation. The new algorithm always performs at least as well as both traditional hash-based and traditional sort-based algorithms. It can serve as a systems only aggregation algorithm for unsorted inputs, thus preventing erroneous algorithm choices. Furthermore, the new algorithm produces sorted output that can speed up subsequent operations. Googles *F1 Query* uses the new algorithm in production workloads that aggregate petabytes of data every day.

PVLDB Reference Format:
Thanh Do, Goetz Graefe. Sort-based grouping and aggregation. *PVLDB*, 12(xxx): xxxx-yyyy, 2021.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

There is a wide variety of algorithms for duplicate removal, e.g., in SQL queries like “select distinct A, B from”. Most of these algorithms are also suitable for grouping and aggregation, e.g., in SQL queries like “select A, B, count (*), sum (X), avg (Y), min (Z) from group by A, B”. If the data in the “from” clause are already sorted on “A, B” or something equivalent, in-stream grouping and aggregation is very simple and very efficient. If the input is sorted on a prefix of the required sort key, e.g., only on “A”, then the algorithms below apply one segment at a time, e.g., for grouping on “B within segments defined by distinct values of “A”. Otherwise, if the output size is such that in-memory computation suffices, avoiding any need for temporary storage on external devices, then the concerns and techniques below apply to data movement between CPU caches and system memory. If the input size and its storage location are such that parallel computation is desirable, partitioning permits local and independent computation of the query result, e.g., partitioning on “hash (A, B). If re-partitioning (shuffle, exchange) is required, best-effort in-memory duplicate removal, grouping and aggregation can reduce the communication effort. What remains is the need for an efficient sequential algorithm for duplicate removal, grouping, and aggregation of large unsorted inputs.

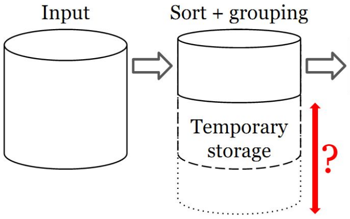

Figure 1: Optimization opportunity in sorting and grouping

Figure 1 illustrates the principal optimization opportunity in a sequential grouping algorithm for unsorted inputs. The input and output may be stored files (as shown) or they may be transient data streams. More importantly here, the sizes of input and output are fixed and their costs cannot be avoided or reduced by optimizing the grouping algorithm. The biggest optimization opportunity within the grouping operation is avoiding or reducing the need for temporary storage. If both input and output are larger than the available memory, pipeline-breaking “stop-and-go algorithms cannot avoid temporary storage altogether. The question is whether requirements for temporary storage equal the output size, equal the input size, or exceed both sizes, e.g., due to multi-level partitioning or merging.

For unsorted inputs, there are two kinds of grouping algorithms. Both are classic divide-and-conquer designs. The first kind of algorithm hash-partitions the data into disjoint subsets, either in memory, usually as buckets in a hash table,

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

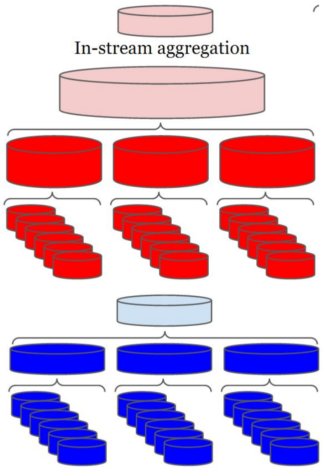

Figure 2: In-stream duplicate removal and in-sort duplicate removal within runs

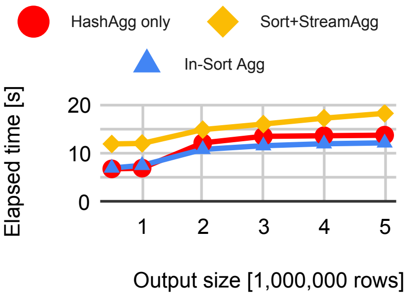

Figure 3: Motivating performance example from Googles *F1 Query*

or on temporary storage, often called partitions or overflow files. The second kind of algorithm sorts the data on all columns (fields, attributes) for duplicate removal or on the grouping columns for grouping and aggregation. The standard sort algorithm is an external merge sort with a variety of optimizations for performance and for graceful degradation, e.g., an incremental transition from in-memory sorting to external sorting. Some implementations employ a mixed approach, e.g., a hash table in memory and merge sort as external algorithm. For example, Boncz et al. [4] mention “hash-based early aggregation in a sort-based spilling approach. Another example for this mixed approach is the initial implementation of duplicate removal, grouping, and aggregation in Google’s *F1 Query* [28, 30].

Sort- and hash-based query processing are more similar than commonly recognized [11]. To wit, Mller et al. [25] offer the insight that “hashing is in fact equivalent to sorting by hash value. They err, however, in “hashing allows for early aggregation while sorting does not. Perhaps they learned this erroneous understanding from [11] and [13]. One of the techniques introduced in the present paper eliminates this misunderstanding.

Sorting and duplicate removal are not new research topics, of course. For example, Hrder [19] summarizes that “eliminating duplicates can be achieved by a single sort (not after a sort). In a footnote, Bitton and DeWitt [3] credit System R (and thus Hrder) with duplicate elimination in intermediate runs. Neither of these papers explicitly mentions the similarity of algorithms for duplicate elimination, for grouping and aggregation, and for minimizing the invocation frequency of expensive operations [20], e.g., of fetching rows by row identifiers, of nested queries, and of user-defined functions.

Figure 2 illustrates duplicate removal within runs [3, 19]. This tiny example assumes that input and memory sizes force 18 initial runs on temporary storage and that memory and page sizes limit the merge fan-in to 6. On the top, after a traditional external merge sort generates and merges runs, it pipelines the output of the final merge step into an in-stream aggregation operation. The result of the sort is just as large as the unsorted input; only the subsequent in-stream aggregation reduces the data volume. On the bottom, duplicate removal within runs reduces the data volume on temporary storage. Intermediate runs are never larger than the final result, which the final merge step computes.

The present paper introduces two new techniques. Both improve external merge sort in the context of duplicate removal, grouping, and aggregation; and both employ in-memory indexes where traditional designs employ priority queues. The first new technique, early aggregation, improves run generation or the input phase of external merge sort. It matches a commonly cited advantage of hash-based duplicate removal, grouping, and aggregation for unsorted input and in-memory results, e.g., for TPC-H Query 1. The second new technique, wide merging, improves the final merge step or the output phase of external merge sort. Together, these two techniques ensure that sort-based duplicate removal, grouping, and aggregation is competitive with hash-based algorithms for any input size and any output size. Of course, sort-based query processing has many other advantages commonly known as “interesting orderings [29]. Many of these advantages also apply to other sort-based dataflow environments, e.g., MapReduce [8] and its many successors.

A single algorithm for duplicate removal, grouping, and aggregation with robust performance (matching the best prior algorithms) is more than an intellectual curiosity for the algorithm enthusiast. In many practical ways, it benefits any production system, not only in terms of code volume and effort for code maintenance but also in terms of query optimization complexity and uncertainty in algorithm choices. Other benefits apply to query execution policies, e.g., the complexity of memory management, and to physical database design, application tuning, data center monitoring, and user education.

In the implementation of Googles *F1 Query* [28, 30], hash join applies recursive partitioning using a sequence of hash functions whereas hash aggregation resolves overflow by external merge sort. Adding hash partitioning to the existing in-memory hash aggregation suggests itself, but it turns out that sort-based duplicate removal, grouping, and aggregation can always be as fast and much faster when interesting orderings [29] matter.

Figure 3 compares the performance of duplicate removal in *F1 Query* for an unsorted input of 6,000,000 rows, memory for 1,000,000 rows, and a variety of output sizes. All algorithms are implemented and tuned for production. A traditional external merge sort with subsequent in-stream aggregation is the most expensive option in all cases. Traditional hash aggregation performs very well if the output fits in memory and degrades somewhat gracefully for outputs larger than memory. In-sort aggregation with the new techniques performs slightly worse than hash aggregation for small outputs and somewhat better than hash aggregation for large outputs. Given competitive performance, more graceful degradation, and the ability to produce output in interesting orderings, in-sort aggregation seems suitable as the only algorithm for duplicate removal, grouping, and aggregation for unsorted inputs.

Among the following sections, the next one reviews related prior work. Section 3 introduces sort-based early aggregation while consuming unsorted inputs, i.e., before in-memory sorting and thus long before writing initial runs to temporary storage [3]. Section 4 introduces wide merging in the final merge step, i.e., with a fan-in much higher than traditional merging. Section 5 provides some background on our product implementation. Section 6 details performance measurements and the final section summarizes and offers a few conclusions.

2. RELATED PRIOR WORK

This section reviews prior work on query processing in relational databases, in particular sorting, hashing, and grouping algorithms.

2.1 Interesting orderings

From early relational database management systems, sort-based algorithms and sort order have been central to query processing engines. Early research into query evaluation and grouping algorithms [10] discussed duplicate removal within sort operations and in-stream grouping for sorted inputs: "...first project the needed domains and then sort on the by-list being careful not to remove duplicates ...Since the tuples are sorted in order of the by-list, each tuple read will have either the same by-list as the previous tuple, or it will be an entirely new by-list and there will be no more references to any previous by-lists." The same research also considered grouping using in-memory hash indexes: "...the assumption that $B \geq P$ [memory size $\geq$ output size] is commonly true in practice. To the extent that this holds, the best structure to use is hash, and sorting does not help. If $B < P$ and U [output row count] is large, then sorting clearly wins."

Early research into query optimization crystallized the concept of interesting orderings and their effect on query evaluation plans [29]. Sort-based algorithms such as merge join have obvious positive interactions with sorted storage structures such as b-tree indexes as well as queries with "order by clauses. Multiple joins on the same primary key and foreign keys are common in re-assembly of complex objects after relational normalization in the database. Grouping on foreign keys is common because it computes an aggregate property of the larger entity, e.g., the total value of all line items in a purchasing order. Thus, grouping operations before or after primary key-foreign key joins are found in many queries and query evaluation plans.

Partitioning is a similar physical property considered during compile-time query optimization and enforced by re-partitioning, also known as exchange or shuffle. Perhaps these two physical properties, sorting and partitioning, are so similar that a single operation should enforce both of them, but we leave this question to future research.

2.2 Applications of sort-based grouping and aggregation

The algorithms discussed in this paper support sort-based duplicate removal, grouping, and aggregation. These discussions go beyond earlier descriptions of sorting and duplicate removal in relational database management systems [3, 13, 19]. A related operation avoids redundant invocations of expensive operations such as (correlated) nested queries and user-defined functions [20].

A typical example of a large duplicate removal operation is counting distinct users in a popular web service. Logs generated by web servers may produce billions of log records per day. A dataflow pipeline or a SQL query extracts user identifiers and then removes duplicates, i.e., multiple log records pertaining to the same user. For a popular web service, this reduces many billions of rows to many millions of rows.

If counts are desired per hour or per country, the required grouping operation can use the same algorithm. In hash-based query processing, one operation (with hash table and hash-partitioning) removes duplicate user ids and another operation (with its own hash table and hash-partitioning) counts users per hour and country. In sort-based query processing, a single sort operation (on country, hour, and user identifier) serves both duplicate removal and subsequent grouping. A related problem not only removes duplicate user identifiers but aggregates multiple records in the web log to a session per user.

"Group-join is a special algorithm feature invented repeatedly for hash join [12, 15, 26]. The innovation is to use the same hash table for both grouping and subsequent join. It is particularly effective when first grouping and then joining on a foreign key. Unfortunately, it applies only to a hash join's build input and thus inhibits role reversal, whereas in sort-based query processing, in-stream grouping naturally applies to both inputs of merge join as well as its output. For unsorted join inputs, the sort logic can apply duplicate removal, grouping, and aggregation.

Rollup functionality has existed for a long time in programming environments such as Cobol and been suggested for database queries [11]. Sort-based aggregation can compute multiple levels of aggregation with a single sort operation, e.g., for a query of the form "select ... group by rollup (day, month, year)". In contrast, hash-based aggregation requires separate computations for each level of aggregation. Each level requires a hash table and possibly hash partitioning to temporary storage.

Log-structured merge-forests and stepped-merge forests [22, 27] are nearly ubiquitous in key-value stores. In this context, runs are often called deltas and merging is often called compaction, because merging includes aspects of aggregation and compression. The individual merge steps are similar to those of external merge sort, but their merge policies (what to merge when) are quite different for multiple reasons. First, their input is assumed endless. For example, it is not possible to delay merging until run generation is complete; merging must be concurrent to run generation. Second, inputs include traditional insertions, which are mapped to append operations, as well as updates, which are mapped to insertions of replacement rows, and deletions, which are mapped to insertions of "tombstone" rows. The merge logic aggregates insertions, updates, and deletions either into a final state or into recent history of versions, including removal of out-of-date versions. Third, individual runs are formatted as b-trees, not flat files, in order to permit search and queries over recent as well as historical information. Alternative formats include a single partitioned b-tree, with runs mapped to partitions. Bit vector filters can enable a query to skip some partitions and thus improve performance. Fourth, the merge fan-in and the frequency of merge step are controlled not by the memory size but by the desire for good query performance, i.e., searching few partitions. Many designs and deployments employ low-fan-in merge steps, even binary merging.

Decades ago, Gray suggested sorting recovery log records on the database page identifier to which they pertain [17]: “This compressed redo log is called a change accumulation log. Since it is sorted by physical address, media recovery becomes a merge of the image dump of the object and its change accumulation tape.” It seems a small step from sorting recovery log records to building b-tree indexes, another step to building indexes incrementally and continuously (in the manner of log-structured merge-forests), and yet another small step to using such indexes for page-by-page incremental, on-demand, “instant” recovery from single-page failures and from media failures [16].

2.3 Optimizing “group by" and “order by" queries

Functional dependencies enable many interesting optimizations for “group by” and “order by” queries [31]. Functional dependencies are implied by primary key integrity constraints and by prior “group by” or “distinct” operations.

More specifically, a “group by” clause requires a set of columns (scalar expressions) and an “order by” clause requires a list of columns. Functionally dependent columns can be removed anywhere in a set and only in subsequent positions within a list. For example, in two database tables for purchase orders and their line items, with o_orderdate functionally dependent on o_orderkey, three among the following four clauses permit simplification but the last one does not:

1. “. . . group by o_orderkey, o_orderdate”,
2. “. . . order by o_orderkey, o_orderdate”,
3. “. . . group by o_orderdate, o_orderkey”,
4. “. . . order by o_orderdate, o_orderkey”.

Below is a (first) example of using functional dependencies in an unusual way. Among the following equivalent queries, changes from variant to variant are underlined. The first query seems to require grouping and aggregation after the join, but the second and third queries are essentially equivalent to the first one due to the functional dependency of order date on the grouping key. Adding a functionally dependent column to a “group by”” clause applies the insights of [31] in the reverse direction. As grouping key and join key are the same, order date is a constant within each group of line items. The fourth query variant is equivalent to the first query and most conducive to efficient execution. Note that the many-to-one join changes into a one-to-one join.

1. “select o_orderkey, avg (l_shipdate o_orderdate)
   from lineitem join orders on (l_orderkey = o_orderkey)
   group by o_orderkey”

2. “select o_orderkey, o_orderdate,
   avg (l_shipdate o_orderdate)
   from lineitem join orders on (l_orderkey = o_orderkey)
   group by o_orderkey, o_orderdate”

3. “select o_orderkey, o_orderdate, avg (l_shipdate) o_orderdate
   from lineitem join orders on (l_orderkey = o_orderkey)
   group by o_orderkey, o_orderdate”

4. “select o_orderkey, avg_shipdate o_orderdate
   from (select l_orderkey, avg (l_shipdate) as avg_shipdate
   from lineitem
   group by l_orderkey) as a
   join orders on (l_orderkey = o_orderkey)”

In many queries, query rewriting such as this example is required to enable “group-join.” In hash-based query processing, optimizing grouping and join on the same column (set) applies only to the build input. In sort-based query processing, grouping and join on the same column (set) enjoy the benefits of interesting orderings if grouping is applied to either of the two join inputs or to the join output. In other words, interesting orderings benefit query performance whether or not query optimization applies all kinds of clever and uncommon rewrites.

In sum, sort-based duplicate removal, grouping, and aggregation can benefit from proper use of functional dependencies because they permit optimizations of both grouping and ordering, but it seems that sort-based query evaluation plans are somewhat more forgiving and flexible than hash-based query execution.

2.4 Optimizations of sort operations

High-performance sorting requires efficiency, scalability, and robustness (of performance). Efficiency may benefit from tree-of-losers priority queues [23], normalized keys, offset-value coding [6], and hardware support, e.g., for tree-of-losers priority queues and offset-value coding in normalized keys [21]. The techniques introduced in Sections 3 and 4 improve the efficiency of sort-based duplicate removal, grouping, and aggregation, specifically the input phase and the output phase of external merge sort in those operations.

Scalability is principally about parallelism twice the resources should process the same data in half the time or twice the data in the same time. Robustness of performance is about performance cliffs and graceful degradation for example, the transition from an in-memory quicksort to an external merge sort should be gradual rather than a binary switch, such that a single additional byte or input row cannot force spilling the entire memory contents. The techniques introduced in Sections 3 and 4 are orthogonal to both scalability and robustness of performance: the new techniques do not offer improvements in those directions but they also do not impede or hinder existing or future techniques for scalability and robustness.

2.5 Early results in join-by-grouping

Complementing applications and optimizations of sort-based grouping, there is an unusual join algorithm based on grouping. It requires that the implementation of external merge sort can interleave multiple record types within memory and within each run on temporary storage. Sorting a mixed stream of records on join key values produces mixed “value packets” [24], i.e., sets of rows with equal sort keys. In the context here, equal sort keys means equal join keys. Forming or combining value packets is a kind of aggregation. The join output is computed from the final sorted stream by computing a cross product within each mixed value packet. Alternatively, when the sort and merge logic forms or combines mixed value packets, it can produce join results as an

A	B
13	22
11	24
17	28
13	25
15	26
13	29

A	C
13	37
12	36
13	34

A	B	C
11	24	
13	22	
13	29	
13		34
15	26	

A	B	C
12		36
13	25	
13		37
17	28	

A	B	C
11	24	
12		36
13	22	
13	25	
13	29	
13		34
13		37
15	26	
17	28	

A	B	C
13	22	37
13	29	37
13	25	34

A	B	C
13	22	34
13	29	34

A	B	C
13	25	37

Figure 4: Join by grouping

immediate side effect. In other words, early aggregation in this context means early and incremental join results. Variants of this algorithm can compute semi-join, anti-semi-join, all forms of outer join, set and bag intersection and difference (e.g., "intersect all" in SQL). Anti-semi-join and equivalent result rows of outer joins cannot be produced early.

Figure 4 illustrates one merge step within this unusual algorithm applied to an inner join. Two unsorted input tables (far left top and bottom) are joined on column A. Rows from both inputs are scanned concurrently and run generation creates initial sorted runs (center left). These runs contain multiple record types, one for each join input. Merging runs (from center left to center right) relies on the standard merge logic known from external merge sort. As a side effect, this merge step combines value packets (in this example, for $A = 13$) and produces new join results (far right top). When run generation assembled the merge inputs (center left), it produced early join results (far right middle and bottom) from value packets in these runs.

Once two records have been joined, they remain in the same value packet until the sort finishes. Hence, there is no danger of duplicate (redundant, wrong) output. For example, in Figure 4, the two original inputs (far left) contain 3 and 2 rows with key value $A = 13$ for $3 \times 2 = 6$ rows in the join result; the three partial results (far right top to bottom) contain precisely these 6 rows. When the sort finishes, the final value packets are dropped; the operations output is the join result computed incrementally as side effect of forming and combining mixed value packets.

This join algorithm is an alternative to more complex sort-based algorithms for joins with early output [9]. Its output rate and memory requirements mirror those of symmetric hash join [32] if early aggregation and wide merging are enabled, which are the topics of the next two sections.

2.6 Summary of related prior work

To summarize our observations on related prior work, duplicate removal, grouping, and aggregation occur in a large variety of contexts, from business intelligence to web logs and recovery logs. Substantial research and development effort have been invested in both query optimization and query execution specifically for duplicate removal, grouping, and aggregation. A remaining thorny problem is that traditional sort- and hash-based algorithms are optimal in different circumstances, rendering a choice during query optimization difficult and error-prone. Instead, the next two sections offer a single algorithm that, assuming equal care in algorithm implementation, always matches the best traditional algorithm for duplicate removal, grouping, and aggregation.

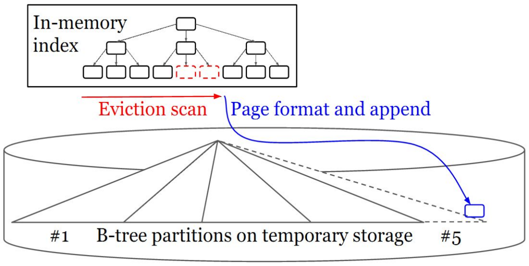

Figure 5: Run generation using an ordered in-memory index

3. EARLY AGGREGATION DURING RUN GENERATION

Techniques for early duplicate removal, grouping, and aggregation are particularly valuable for queries with small results, i.e., duplicate removal or aggregation with many input rows and few output rows. More specifically, if the output fits in the memory allocation available for the grouping operation but the input is unsorted and large (such that expensive spilling to temporary storage is required in a traditional sort algorithm), then early aggregation improves the performance of sort-based aggregation. In fact, early aggregation ensures that sort-based aggregation spills no more data to temporary storage than hash-based aggregation and sometimes a little bit less.

Early aggregation pertains to the input phase of an external merge sort, i.e., run generation. Traditional run generation employs read-sort-write cycles or replacement selection. The former uses an internal sort algorithm such as quicksort for initial runs as large as memory; the latter uses a priority queue and can produce initial runs twice as large as memory for truly random input, as large as memory in the worst case, and as large as the entire input in the very best case.

In contrast, early aggregation eschews both quicksort and priority queues; instead, it uses an ordered in-memory index. Both read-sort-write cycles and replacement selection are possible. The index enables immediate discovery of duplicate key value, just like a hash table. In fact, if the sort key is a hash value, a hash table can serve as the ordered index. If the output size is smaller than the memory size, early aggregation avoids all I/O to spill intermediate data into runs on temporary storage.

Figure 5 illustrates run generation using an ordered in-memory index. The index continuously grows due to insertions of rows and key values from the unsorted input. New key values create new index entries; key values equal to ones already in the index are absorbed by aggregation. In the ideal case, the entire input can be absorbed within

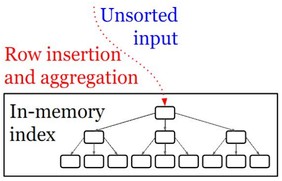

Figure 6: In-memory aggregation

memory. Otherwise, a scan thread continuously evicts leaf pages and writes them to runs on temporary storage, where they form a forest (of many trees) or a partitioned b-tree (of many partitions).

3.1 Example 1

As an archetypal example of a grouping query with a small output, consider Query 1 of the TPC-H benchmark [1]. In scale factor $SF = 1$ of the benchmark, the query scans a database table of about 6 M rows, selects about 90% of those, and then reduces them to four rows with counts, sums, and averages. For scale factor $SF = 1,000$, the query reduces $6B$ rows to four output rows. Clearly, an external merge sort of $6M$ or even $6B$ rows with subsequent grouping and aggregation, i.e., the two techniques of Figure 2, cannot compete with an algorithm that uses a hash table to simply accumulate the final query result within a small memory allocation of 4 rows.

There are many real-world queries in which a grouping result fits within the readily available memory allocation. For example, if each operator within a query evaluation plan is allotted 100 MB by default, then any grouping operation with a result smaller than 100 MB should remain an in-memory operation and algorithm. This is, of course, true for hash aggregation but it can also be true for sort-based duplicate removal, grouping, and aggregation.

Figure 6 illustrates this case. The in-memory index can grow until it fills memory. Skew in the key value distribution does not matter as an ordered index adapts to the actual distribution. Only if the output size exceeds the available memory, spilling to runs on temporary storage as shown in Figure 5 is required.

3.2 Example 2

As another example, imagine the "group by" clause of Example 1 extended such that the final output is larger than memory, i.e., $O > M$ or more specifically $O = 2M$. Even with early duplicate removal, grouping, and aggregation, this example requires runs on temporary storage. As key values in the in-memory index are unique, the in-memory index immediately matches and absorbs the fraction of the input rows. With run generation by replacement selection and memory always full, about $M/O = \frac{1}{2}$ of all input rows are absorbed immediately. Ignoring the effects of an in-memory run for graceful degradation from in-memory sorting to external merge sort, the total size of all initial runs is about $M + (1M/O) * I$ or in the specific example $M + \frac{1}{2}I$. With only unique key values in the in-memory index, the traditional logic for duplicate removal, grouping, and aggregation [3] while writing runs on temporary storage is not required.

Figure 7 shows the predicted spill volume for input size $I = 1,000,000$ rows and memory size $M = 100,000$ rows. If the output size equals the memory size ($O = M$), no spilling is required. If the output is many times larger than the available memory allocation, practically all input rows spill. The calculation in this prediction assumes replacement selection using an in-memory index, even though our implementation uses read-sort-write cycles. (Recall that replacement selection is usually implemented using a priority queue and for random input produces runs twice the size of memory, whereas read-sort-write cycles are usually implemented using quicksort and produces runs the size of memory; recall also that an ordered in-memory index permits either read-sort-write cycles or replacement selection.)

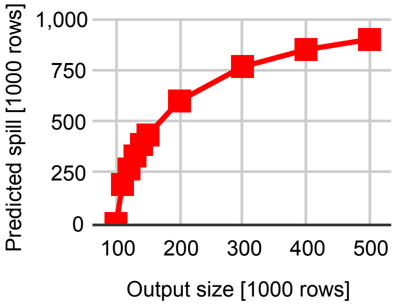

Figure 7: Predicted spill volume

3.3 Algorithms and data structures

One design combining in-memory run generation with early aggregation uses two data structures. For example, Boncz et al. [4] mention "hash-based early aggregation in a sort-based spilling approach." If the in-memory hash table fails to absorb (i.e., to aggregate) an input row, the row is added to the in-memory data structure, whether that is an array for quicksort or a priority queue for replacement selection.

An alternative design employs a single data structure for two purposes, searching and sorting. A suitable data structure is an ordered in-memory index, e.g., an in-memory b-tree [2, 14]. Note that comparisons are required only during the search. If no match is found, the search for a match has already found the insertion point as a side effect.

If the search employs a binary search within each tree node, the count of comparisons per input row is equal to that in a tree-of-losers priority queue, which is 10-20% lower than the count of comparisons in quicksort and very close to the theoretical minimum. (Sorting N items is equivalent to selecting one of N! permutations, which requires $\log_2(N!)$ comparisons.)

Interpolation search within each tree node is even more efficient if the key value distribution is nearly uniform, which is likely the case if the sort key is a hash value. Note that sorting on hash values permits exploiting interesting orderings if other algorithms and storage structures also sort on hash values. Merge joins and b-trees on hash values are attractive for database columns with no real-world "<" comparison, e.g., practically all artificial identifiers and thus many primary keys and foreign keys in real-world databases.

The row format within the in-memory index is the same as in runs on temporary storage. It may differ from the row formats in both the input and the output. For example, in addition to a grouping key, input rows may contain a value, intermediate rows a sum and a count, and output rows an average. Similar considerations apply when computing variance, standard deviation, co-variance, correlation, regression slope and intercept, etc.

In traditional run generation, read-sort-write cycles may use quicksort to produce runs the size of memory. Run generation by replacement selection using a priority queue can produce runs the size of memory or, with an additional comparison for each new input row as well as a flag within each row in memory, twice the size of memory. Run generation using an in-memory index can produce runs twice the size of memory without an additional comparison and without a flag in each row in memory. Eviction from memory to temporary storage repeatedly scans the in-memory index as shown in Figure 5. Advancing the scan evicts rows and frees index nodes whenever the in-memory index needs to split a node and thus allocate a free node. An alternative design uses a partitioned b-tree in memory, with partitions matching runs on temporary storage. In this design, the eviction scan moves the left edge of the b-tree. The current key value of the eviction scan governs assignment of new input rows to partitions and runs. In yet another variant, if new input rows are always assigned to the latest partition in memory, runs on temporary storage will be the size of memory.

3.4 Search optimizations

Since each input row requires a search within the in-memory index, optimizing the search logic is crucial. We have used the following (well-known) optimizations and observed performance advantages:

First, nodes in memory can be large. Thus, the b-tree in memory is quite shallow. For example, a memory allocation of 100 MB permits 1 M rows of 100 bytes. With a key size of 10 bytes and 70% space utilization in all nodes, a node size of 150 KB permits a b-tree without intermediate levels and a node size of 8 KB permits a b-tree with a single intermediate level. Note that a hash table also requires either large contiguous memory or a tree structure. In our context, we have found that interpolation search and hardware-assistance for search can be quite effective, and that both are aided by large node size.

Second, instead of a root-to-leaf search in the b-tree for each input row, small batches of search keys turn a search problem into a merge problem. The key values in each batch are sorted before searching in the b-tree. While processing the batch, each search starts in the leaf node located for the preceding key value within the batch. If the count of distinct key values in a batch exceeds the count of leaf nodes in the b-tree, this strategy can be very efficient.

3.5 Analysis

Two questions suggest themselves:

1. How many comparisons are required in early aggregation, i.e., run generation with an in-memory index? How does that compare to run generation with read-sort-write cycles, e.g., quicksort, and with replacement selection, i.e., a tree-of-losers priority queue?
2. If the output size is a small multiple of the memory size, what fraction of input rows are absorbed immediately in the in-memory index, and how many rows are spilled to runs on temporary storage?

The count of comparisons per input record in run generation using an efficient (tree-of-losers) priority queue is $\log_2(M/R)$ for memory size M and record size R. This is correct for run size equal to memory size M; one additional comparison is required for expected run size $2M$. Using quicksort, the expected count of comparisons is 10-20% higher; the worst case is much higher. In run generation using an in-memory index with binary search, the count of comparisons per input record is again $\log_2(M/R)$. Using interpolation search, it can be substantially lower.

If the final output fits in memory, i.e., $O < M$, then the count of comparisons per input record is $\log_2(O/R)$. It can be substantially lower with interpolation search. Linear interpolation is effective if the key value distribution in the output is practically uniform. This is probable if the keys are hash values, i.e., when sorting and grouping on hash values, at least as the leading sort key.

In a striking similarity, hash aggregation requires a search in the hash table for each input record, i.e., a hash calculation plus a scan through a hash bucket. Those are comparable to the interpolation calculation and the local search near the interpolated position. If memory remains full all the time during run generation, then each input row has a probability of M/O (memory size over output size) of finding a matching key value in memory and of being absorbed in the index without insertion. If $M \geq O$, this probability is a certainty and spilling to runs on temporary storage is not required. If this probability is very small, then practically all input rows spill into runs on temporary storage.

3.6 Summary of early aggregation

To summarize, early aggregation uses an in-memory index to match and absorb input rows during run generation in a duplicate removal, grouping, and aggregation operation. In the ideal case, it entirely avoids spilling rows to runs on temporary storage. A typical example is TPC-H Query 1 with only 4 output rows even for very large databases and input tables.

The in-memory index can be a simple b-tree or it can be optimized in many ways. A binary search guarantees $\log_2(M)$ comparisons per input row. Replacement selection can produce runs twice as large as memory. Whatever the algorithm for run generation, runs require merging with the traditional merge logic known from external merge sort or, in many cases, an optimized merge logic to be known as wide merging.

4. WIDE MERGING IN THE FINAL MERGE STEP

For complete performance parity with hash aggregation, in-sort grouping and duplicate removal requires a final merge step with a merge fan-in potentially higher than a traditional merge step with the same memory allocation and page size (unit of I/O).

Compared to traditional merging in an external merge sort, wide merging is not limited to a specific fan-in. Instead, wide merging uses its memory allocation for an in-memory index and processes one page at a time from the runs of the aggregation input. Thus, wide merging can be much more efficient than traditional aggregation within sort, e.g., saving an entire intermediate merge level.

Figure 8 extends the example of Figure 2 with a third merge strategy for sort-based duplicate removal, grouping, and aggregation from unsorted input data. Wide merging can consume and aggregate many more runs than it has memory pages, e.g., 18 runs with only 6 memory pages. This

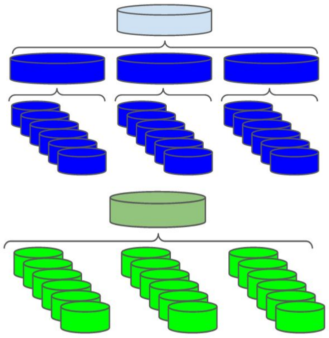

Figure 8: The effect of wide merging

is possible if memory can hold and index a key range equal to the key range of one page in the merge inputs on temporary storage. Wide merging uses only one input buffer shared by all runs. After reading a page, wide merging copies the page contents into its in-memory index before reading the next page, typically from a different input run.

Figure 9 illustrates the flow of data in wide merging. Using a priority queue, the algorithm chooses an input run from which to read the next page. Once that page is in the input buffer, the key values on that page are found in the in-memory index. If a key value exists, the input row is absorbed without growing the index. If the key value does not exist, a new entry is required in the index. In this way, the ordered in-memory index absorbs all rows and key values from all runs or partitions on a temporary storage device. As the merge logic progresses through the domain of key values, the active key range in the in-memory index turns over continuously. The left edge of the in-memory index produces final output and the right edge adds new key values from the merge inputs.

A priority queue guides the page consumption sequence during wide merging. It has an entry for each of its input runs, e.g., 18 entries in the example of Figure 8. It is similar to the priority queue used in traditional external merge sort for guiding read-ahead within the merge input, also known as forecasting [23]. From each of the input runs, it tracks the highest key value already read; the next read operation targets the run with the lowest of these key values. If implemented as a tree-of-losers priority queue, one leaf-to-root traversal in the priority queue is required for each page in the runs on temporary storage, or $\log_2(18) \approx 4$ comparisons in the example of Figure 8. The highest key value in the first page of each run initializes this priority queue. Alternatively, if the sort operation retains minimum and maximum key values of each run, e.g., for the purpose of concatenating runs with disjoint key ranges [19, 13], the retained minimum key values initialize this priority queue.

4.1 Example 3

Consider a specific example of wide merging and its benefits: single-threaded duplicate removal with input size $I = 750,000$ rows, memory size $M = 1,000$ rows, merge fan-in (in traditional merge logic) and partitioning fan-out $F = 6$, and final output size $O = 32,000$ rows. Importantly, the memory is much smaller than the final output and the final output is much smaller than the original input, or $M \ll O \ll I$.

In this example, hash aggregation invokes $L = 2$ partitioning levels to divide all input rows into $F^2 = 36$ partitions of about $I/F^2 = 21,000$ rows each. During these partitioning steps, the output buffers are too small to enable much early (opportunistic) duplicate removal, grouping, and aggregation. After these partitioning levels, each partition contributes about O/F^2 rows to the final output. As the output per partition is smaller than memory ($O/F^2 = 900 < 1,000 = M$), grouping and aggregation can occur in memory in spite of input partitions much larger than memory ($I/F^2 = 21,000 > 1,000 = M$). The total size of all temporary partitions is $I \times 2 = 1,500,000$ rows, each written and read once. More generally, hash aggregation can aggregate the remaining partitions in memory after $L \geq \log_F (O/M)$ partitioning levels, which is here $\log_6(32) = 2$ partitioning levels.

In contrast, in-sort duplicate removal, grouping, and aggregation starts with run generation by replacement selection. Each run holds about $2M = 2,000$ rows; thus, this example requires about $I/(2M) = 376$ runs. The first merge level produces $376/F = 63$ runs of about $2,000 \times F = 12,000$ rows. The second merge level produces $63/F = 11$ runs. Aggregation within runs [3] cuts their size from $6 \times 12,000 = 72,000$ rows to $O = 32,000$ rows. The penultimate merge step combines 6 of these 11 runs into another run of $O = 32,000$ rows and the last merge step produces the final output. The total size of all runs spilled to temporary storage during run generation, full merge, optimized merge, and partial merge is $750,000 + 750,000 + 11 \times 32,000 + 32,000 = 1,884,000$ rows, each written and read once. This is about 25% more than the temporary partitions in hash aggregation.

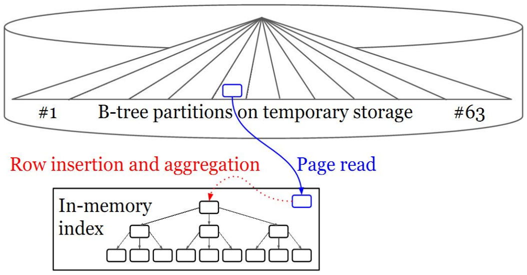
#1
B-tree partitions on temporary storage
#63
Row insertion and aggregation
Page read
In-memory
index

Figure 9: Wide merging using an ordered in-memory index

Wide merging enables further savings. In this example, a single final merge step can aggregate the 63 runs after the first merge level. Merging 63 runs with memory for only a few input buffers requires an in-memory index for immediate grouping and aggregation.

A traditional merge step merging 6 of these 63 runs has its output cut to $O = 32,000$ rows. This is true whether the merge logic uses traditional single-page buffers and a priority queue or an in-memory index for wide merging. If such an index holds practically all distinct key values over the course of the merge step, it can (with the appropriate timing and I/O schedule) absorb rows and key values not only from 6 but any number of runs, e.g., all 63 runs in this example.

With wide merging, i.e., the final merge step immediately after run generation and one full merge level, the total size of all temporary runs is $750,000+750,000 = 1,500,000$ rows and thus perfectly competitive with hash aggregation. As in the cost calculation for hash aggregation, the size of the original input determines the cost of each partitioning or

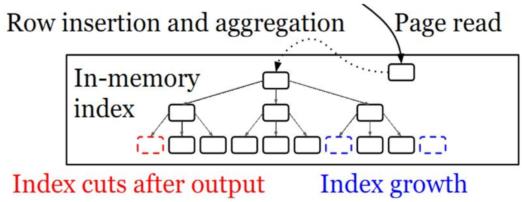

Figure 10: Index expansion and contraction during wide merging

merge level yet the size of the final output (together with memory size and partitioning fan-out or merge fan-in) determines the count of required partitioning or merge levels.

Wide merging with duplicate removal, grouping, and aggregation using an in-memory index proceeds in O/M steps. Each step produces a memory load of candidate output rows, with gradual progression from one step to the next. The runs being merged must have more than O/M data pages such that the key range of each data page is no larger than the key range of the in-memory index. After run generation, in each row's first temporary run on storage, the size of runs is M or 2M. In each merge level, the size increases by the fan-in F. The count of merge levels L must ensure that $F^L \geq O/M$ or $L \geq \log_F(O/M)$. The similarity to the expression for partitioning levels required in hash aggregation is striking.

Figure 10 illustrates this gradual progress through the key domain as required for wide merging as shown in Figure 9. A priority queue guides the page reads. For key values earlier than the top value of the priority queue, aggregates such as counts and sums are final. These can be produced as final output and removed from the in-memory index. As additional pages are read and their key values are inserted, the in-memory index grows, mostly on the right edge but occasionally also elsewhere.

4.2 Example 4

Perhaps a similar example might clarify further. Compared to Example 3, this example uses more realistic values for memory size, merge fan-in, and partitioning fan-out: duplicate removal with input size (per thread) $I = 100,000,000$ rows, memory size (per thread) $M = 100,000$ rows, (traditional) merge fan-in and partitioning fan-out $F = 100$, and final output size (per thread) $O = 8,000,000$ rows. Despite more realistic values for M and F (than used in Example 3), it remains true that $M < O < I$.

Hash aggregation starts with a full partitioning level, which produces 100 partitions of about 1,000,000 rows each. Each partition contributes about 80,000 rows to the final output, which can be aggregated entirely in memory. The total spill volume (partitions on temporary storage) is equal to the input size or 100,000,000 rows. The algorithm requires practically the entire available memory throughout, i.e., during both partitioning and in-memory aggregation. Actually, 90% of memory would suffice: memory size $M = 90,000$ rows and partitioning fan-out $F = 90$ suffice if the hash value distribution is perfectly uniform.

In-sort duplicate removal, grouping, and aggregation starts with run generation, which produces about 500 runs of about 200,000 rows each. Traditional merging requires 5 intermediate merge steps to reduce the count of runs from 500 to 100 as required for the final merge step. Of these, the first merge step uses fan-in 5 and the other ones use fan-in 100. The output size of these steps is limited to the final output size $O = 8,000,000$ rows. Thus, the total spill volume is $I + 1 \times 1,000,000 + 4 \times 8,000,000 = 133,000,000$ rows, which is 33% worse than the total spill volume in hash aggregation. Nonetheless, the algorithm requires the entire available memory throughout, i.e., during run generation, intermediate merge steps, and the final merge step.

In contrast, wide merging can merge and aggregate the initial 500 runs in a single step. Thus, the spill volume is equal to the input size or 100,000,000 rows, matching the performance of hash aggregation. In this algorithm, run generation requires the entire available memory (M = 100,000 rows) but the final (wide) merge step and its in-memory index require only about 40,000 rows (40% of M).

Alternatively, a memory allocation for run generation of $M = 64,000$ rows reduces the size of initial runs to about 128,000 rows and produces about 782 runs. Wide merging can combine all these runs in a single final merge step using a memory allocation of about 63,000 rows. In other words, wide merging permits sort-based duplicate removal, grouping, and aggregation with equal I/O volume but with a lower memory allocation than hash aggregation.

4.3 Analysis

A few questions suggest themselves for further analysis.

1. Is wide merging or an in-memory index useful in external merge sort without duplicate removal, grouping, and aggregation?
2. How many traditional merge levels are required before wide merging applies and becomes effective?
3. Should those earlier merge levels use priority queues (like traditional external merge sort) or should they use an in-memory index (like wide merging)?
4. What is the relationship between traditional early aggregation [3] and wide merging?
5. For quickest application of wide merging, what policy should guide early merging in external merge sort for duplicate removal, grouping, and aggregation?

In response, it seems that run generation and final merge step using an in-memory index offer performance advantages only in queries that require duplicate removal, grouping, or aggregation. Ordinary sorting, e.g., for "order by" queries and for merge join operations, does just as well with traditional algorithms and data structures, e.g., quicksort or tree-of-losers priority queues.

Wide merging is useful only in the final merge step; it might require earlier merge levels like traditional external merge sort. The number of traditional merge levels is a function of initial run size (memory size), merge fan-in, and final output size. All merge steps in which the steps total input size is smaller than the operations final output size should use traditional merge logic. This analysis assumes that a merge steps individual inputs are of similar size.

Wide merging applies when merging with a traditional fan-in processes effectively all distinct key values, i.e., when the total merge input is larger than the operations final output. This is precisely the first merge step (or merge level) in which traditional early aggregation [3] first becomes effective. The difference is that wide merging immediately produces the operations final output by consuming all remaining runs, whereas traditional early aggregation still might require multiple merge steps and levels.

If traditional merge steps are required prior to wide merging in sort-based duplicate removal, grouping, and aggregation, these merge steps must create runs at least as large as the operations final output divided by the fan-in of traditional merge steps (O/F). It appears that there is little benefit in creating larger intermediate runs. Runs of size O/F enable traditional early aggregation [3] and, better yet, wide merging. In other words, wide merging replaces (rather than augments) traditional early aggregation. Creating runs of size O/F requires $\log_F(O/M) - 1$ merge levels after run generation creates runs of memory size M. With the final merge step (merge level) using wide merging, sort-based duplicate removal, grouping, and aggregation requires $\log_F(O/M)$ merge levels.

4.4 Combining early aggregation and wide merging

If the final output is only somewhat larger than the available memory, e.g., $O = 2M$ or $O = 3M$, early aggregation during run generation and its in-memory index can absorb some of the input rows without growing the index or spilling rows from memory to runs on temporary storage. For example, if $O = 2M$, the rows in memory can absorb half of all input rows; if $O = 3M$, the in-memory index matches and absorbs a third of all input rows; etc. Nonetheless, a large input can force many runs on temporary storage. In those cases, wide merging can eliminate one or even two merge levels. In other words, a single sort can benefit from both early aggregation and wide merging. With those two techniques and their combined effects, sort-based duplicate removal, grouping, and aggregation always performs very similarly to hash-based alternatives, as discussed further in Section 6.

4.5 Example 5

For another example that differs from Example 4 only in the final output size, consider duplicate removal with input size (per thread) I = 100,000,000 rows, memory size (per thread) M = 100,000 rows, (traditional) merge fan-in and partitioning fan-out F = 100, and final output size (per thread) O = 150,000 rows. In other words, $I \gg O = 1\frac{1}{2}M$.

In hash aggregation with hash-partitioning, about half of all input rows find a match in memory: hybrid hashing is quite effective in this case. However, the total spill volume is about $\frac{1}{2}I = 50,000,000$ rows. Sort-based aggregation with early aggregation matches the same fraction of input rows during creation of the initial runs. With replacement selection and a run size of about $2M = 200,000$ rows, about 250 initial runs are required. With run generation in read-sort-write cycles and a run size of $M = 100,000$ rows, about 500 initial runs are required. This is too much for traditional merging with fan-in $F = 100$, but nonetheless wide merging can finish the aggregation in a single merge step. Thus, this example benefits from both early aggregation and wide merging; with these techniques, sort-based aggregation can match the spill volume and performance of hash aggregation.

4.6 Summary of wide merging

To summarize, wide merging uses its in-memory index and a single input buffer for all runs on temporary storage. It enables the final merge step in duplicate removal, grouping, and aggregation to consume and to combine many more runs than a traditional merge step using an input buffer for each run. Wide merging applies when traditional merging would produce runs larger than the final output of the grouping operation. Matching the performance and I/O volume of hash aggregation in all cases requires both early aggregation and wide merging.

5. PRODUCT IMPLEMENTATION

This section briefly summarizes the implementation of the new in-sort aggregation operator for Google's *F1 Query* [28, 30]. *F1 Query* is a federated query processing platform that executes SQL queries against data stored in different storage systems at Google, e.g., BigTable [5], Spanner [7], Mesa [18], and more. Before this work, *F1 Query* had two aggregation operators. First, for sorted input, in-stream aggregation requires little CPU effort and hardly any memory. The *F1 Query* optimizer chooses in-stream aggregation whenever possible due to interesting orders [29]. Second, for unsorted input, hash-based aggregation relies on an in-memory hash table. This hash-based operator relies on external merge sort when the output is larger than the available memory allocation.

The new in-sort aggregation algorithm reuses the row-plus-row accumulation component of hash-based and in-stream aggregation. A new order-based indexing component is used for detecting duplicates and groups. To achieve competitive performance, we extended existing b-tree code to implement search optimizations described in Section 3.4. For each input batch, the operator first sorts the batch, usually within the cache line as these batches are small, to detect duplicates within a batch. Only distinct key values within a batch are looked up in the ordered index with the guided search technique. When running out of memory, the new operator uses the ordered index to guide the sequence of rows spilled to intermediate storage, creating sorted runs. These runs are eventually merged and aggregated using wide merging. Contrary to our initial design, the current implementation uses read-sort-write cycles, not replacement selection; therefore, the size of initial runs equal the memory size, even if each run might have absorbed substantially more input with replacement selection in duplicate removal and grouping.

As the new in-sort aggregation produces sorted output as a byproduct of using the ordered index, we take advantage of this property in the optimizer. For example, in aggregation queries with a "group by" clause followed by an "order by" clause with the sort keys matching the grouping keys, the *F1 Query* optimizer avoids redundant sorting. Before our new operator, *F1 Query* planned such aggregation queries using either a hash aggregation followed by a sort or a sort followed by an in-stream aggregation. Plan choices can be suboptimal due to missing or inaccurate cardinality information. The new in-sort aggregation operator allows overcoming this problem by always enabling the optimal plan.

6. PERFORMANCE EVALUATION

The present section reports on the performance of the new sort-based grouping algorithm in *F1 Query*. With no innovation in parallel query execution, all experiments here report local or single-threaded efficiency, scalability, and robustness or reliability of performance. There are four groups of performance results. The first group of experiments replicates earlier examples. The second group focuses on early aggregation using an in-memory index for run generation. The

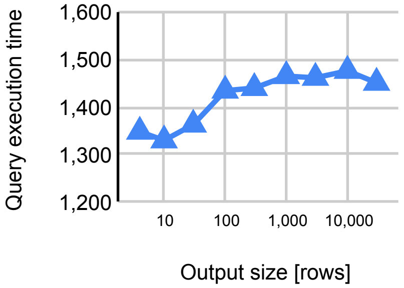

Figure 11: In-memory grouping using a b-tree index

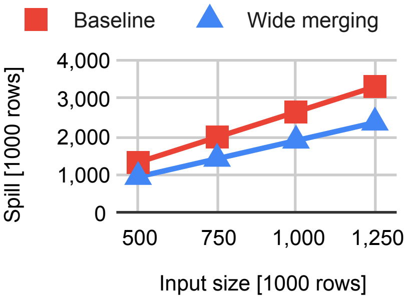

Figure 13: Multiple merge levels

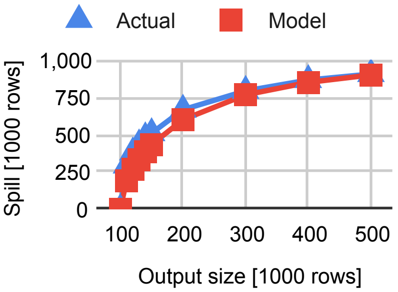

Figure 12: Spill volume to runs on temp. storage

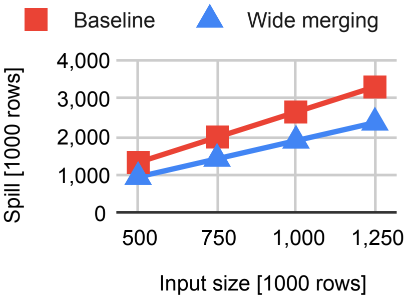

Figure 14: Effect of wide merging

third group focuses on wide merging using an in-memory index during the final merge step. The fourth group of performance results replicate and augment an earlier comparison of sort- and hash-based duplicate removal, grouping, and aggregation. All experiments below ran on a workstation with a local storage device; details are omitted on purpose.

6.1 Validation of examples

Example 1 (Section 3.1) focuses on TPC-H Q1, i.e., a grouping query with a final output smaller than the available memory allocation ($O \leq M$). Figure 11 shows the performance of in-memory grouping and aggregation using an in-memory b-tree index. From left to right, the output size varies from 4 to 30,000 rows. The input size is constant 6,000,000 rows. As is readily apparent, the CPU effort is low and fairly consistent, because any effects due to the logarithmic depth of the ordered index vanishes compared to other CPU efforts in the query evaluation plan.

Example 2 (Section 3.2) also varies TPC-H Q1 with output sizes beyond memory size ($O > M$ but $O < F \times M$). Figure 12 compares the total size of initial runs to a model. The model assumes run generation with replacement selection and computes the spill volume as $M + (1M/O) \times I$. In contrast, our implementation relies on run generation by read-sort-write cycles. Given this difference, the distance between these curves seems acceptable.

Example 3 (Section 4.1) assumes tiny memory, merge fan-in, and partitioning fan-out. Therefore, all algorithms incur multiple partitioning or merge levels. In contrast to traditional sorting and merging, wide merging limits the merge depth $\log_F(O/M)$ versus $\log_F(I/M)$. Figure 13 shows the performance difference between aggregation while writing runs [3] and wide merging: entire merge levels can be avoided, whereas earlier method merely reduce the size of intermediate runs on temporary storage.

Example 4 (Section 4.2) assumes realistic memory size, merge fan-in, and partitioning fan-out. Therefore, a single level of merging suffices if wide merging is available. Figure 14 shows that even for modest input sizes, traditional early aggregation requires multiple merge levels: an algorithm can spill more rows than its input size only if it spills some rows multiple times. In contrast, each input row spills only once when wide merging combines all runs in the first (and last) merge step immediately after run generation.

Example 5 (Section 4.5) shows that in some cases, early aggregation and wide merging are both required for best performance of sort-based duplicate removal, grouping, and aggregation. The experiment in Figure 18 (Section 6.3) confirms the example calculations.

In summary, the examples and the related experiments demonstrate that early aggregation and wide merging, by using in-memory ordered indexes instead of the traditional priority queues, derive substantial benefits for duplicate removal, grouping, and aggregation.

6.2 Early aggregation during run generation

The next experiments and diagrams focus on the hypotheses that, for any input size, output size, row size, page size, and memory size,

1. an ordered in-memory index can be as efficient as a hash table;
2. an in-memory index permits run generation as efficient as quicksort and priority queues; and
3. requirements for temporary storage are the same for an ordered index and run generation as for hash table and hash partitioning.

The experiments cannot claim to cover all sizes and key values distributions, but they may nonetheless helpful in understanding the performance and scalability of in-sort aggregation with early aggregation during run generation.

Figure 15 shows the performance of in-memory aggregation using either a hash table (hash aggregation) or a b-tree (in-sort aggregation). None of these experiments spill to temporary storage. As is readily apparent, the performance of hash table and in-memory b-tree, both properly

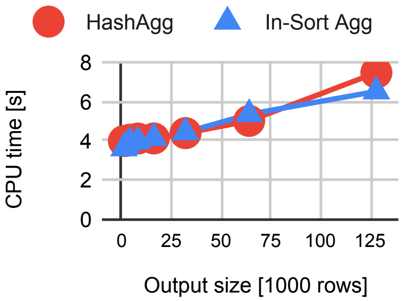

Figure 15: Performance of in-memory indexes

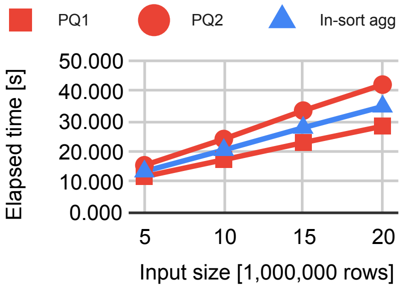

Figure 16: Performance of run generation. "PQ1" and "PQ2" are row-oriented and columnar implementations of the sort operator, respectively.

optimized, is quite similar. Other query execution costs such as predicate evaluation easily dominate their minor differences. In that sense, in-memory b-trees can be just as fast as hash tables, supporting Hypothesis 1.

Figure 16 shows the performance of three implementations of run generation. Two of these use tree-of-losers priority queues; one of them is optimized with normalized keys for fast comparisons and poor man's normalized keys for cache efficiency [13]; the other priority queue is optimized with offset-value coding [6]. The third implementation of run generation uses an in-memory b-tree. Again, other query execution costs easily dominate these differences and the experiment supports Hypothesis 2.

Figure 17 shows the number of runs spilled to temporary storage, including both initial runs. Each run is the size of memory, either a hash table sorted and written as run or a b-tree written in total when memory is full. Recall that the hash aggregation in *F1 Query* uses what Boncz et al. [4] call "hash-based early aggregation in a sort-based spilling approach," which sorts rows in an overflowing hash table, writes them as initial runs on temporary storage, merges those runs, and applies duplicate removal, grouping, and aggregation only during the final merge step. Recall also that our implementation of in-sort aggregation uses its in-memory index for run generation in read-sort-write cycles, not replacement selection. Thus, the counts of initial runs are practically equal, supporting Hypothesis 3.

6.3 Wide merging in the final merge step

The next experiments test the hypotheses that, for any input size, output size, row size, page size, and memory size,

4. wide merging combines many more runs than traditional merging and thus can avoid entire merge levels from traditional sort-based algorithms for duplicate removal, grouping, and aggregation;

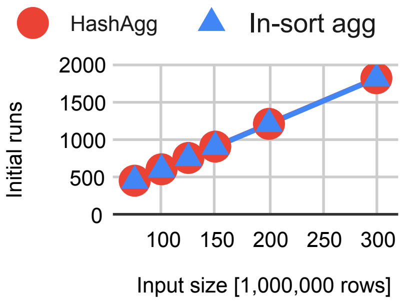

Figure 17: Count of runs spilled from memory to temporary storage

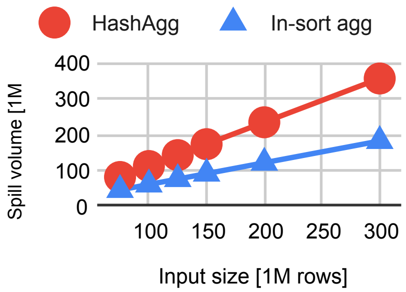

Figure 18: Spill volume.

5. sorting after aggregation can be as expensive as the aggregation such that in cases of equal "group by" and "order by" lists, sort-based aggregation can cost half of hash aggregation plus sorting; and
6. sort-based aggregation can process a "count (A), count (distinct A)" query with grouping in a single sort using both early aggregation and wide merging, whereas hash-based query processing requires two hash aggregation operations the performance difference can equal a factor two.

Figure 18 reports on the total size of all runs for the experiment of Figure 17. Note that this experiment compares optimized in-sort aggregation with the original hash aggregation of *F1 Query*. Both algorithms spill from memory to sorted runs on temporary storage. The two algorithms achieve the same amount of in-memory aggregation during this phase, and thus merging in the two algorithms starts with the same counts and sizes of partially aggregated runs. The original algorithm of *F1 Query* relies on traditional merging, which requires multiple merge steps with intermediate merge results. Thus, the total spill volume exceeds the input size for all input sizes in Figure 18. In contrast, the new algorithm employs wide merging for duplicate removal, grouping, and aggregation. A single merge step suffices and no intermediate merge steps create any additional spill volume. For all input sizes in Figure 18, the total spill volume is much less than the input. One of the data points precisely matches Example 5 (Section 4.5) and all data points support Hypothesis 4.

Figure 19 shows the cost of a query with matching "group by" and "order by" clauses over a table of $I = 6,000,000$ rows. If the output of an initial duplicate removal is small, in particular no larger than memory $M = 1,000,000$ rows, the cost of a subsequent sort operation barely matters. If, however, the intermediate result is large, then satisfying both clauses with a single operation is very beneficial, supporting Hypothesis 5.

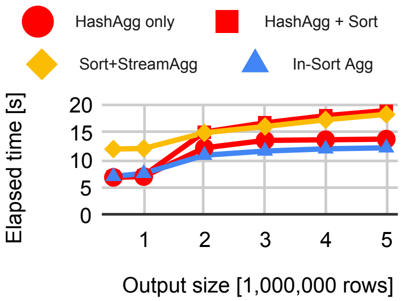

Figure 19: Cost of sorting after aggregation.

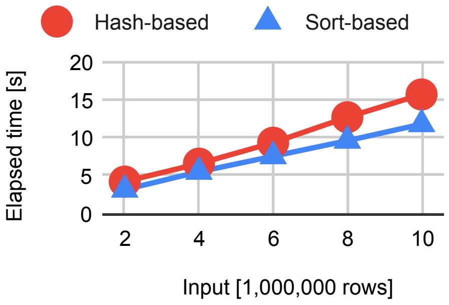

Figure 20: Cost of "count" and "count distinct" queries.

Figure 20 shows the cost of duplicate removal with subsequent grouping. With hash-based algorithms, two hash tables (and possibly overflow to temporary storage) are required. With a sort-based algorithm, a single sort can perform the duplicate removal using an interesting ordering for the subsequent grouping. Thus, only one memory-intensive operation is required with savings up to a factor of two, supporting Hypothesis 6.

6.4 Effects of interesting orders

The next experiment tests the hypotheses that:

7. interesting orderings are important not only for b-tree scans and merge joins but also for query evaluation plans with duplicate removal, grouping, and aggregation;
8. SQL set operations such as "intersect" can be much faster using sort-based query plans than using hash-based query plans; and
9. set intersection and its most efficient query evaluation plans benefit not only users' "intersect" queries but also star queries and snowflake queries in relational data warehousing.

Figure 21 shows a query evaluation plan for a very simple SQL query computing the intersection of two tables, e.g., "select B from T1 intersect select B from T2". If column B is not a primary key in tables T1 and T2, correct execution requires duplicate removal plus a join algorithm. If this is a merge join, the two required sort operations can provide duplicate-free inputs.

Figure 22 shows the performance of sort- and hash-based plans for this query. Each input table has $I = 100,000,000$ rows; the memory for each operator is $M = 10,000,000$ rows. In a hash-based plan, both duplicate removal operations and the join might spill to temporary storage; each input row is spilled twice. In contrast, a sort-based plan spills each input row only once. Thus, the effort for spilling is cut in half due to interesting orderings. Even a group-join, i.e., a hash join supports duplicate removal, grouping, and aggregation on its build input [12, 15], can eliminate only half of this difference.

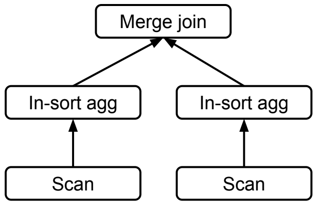

Figure 21: Sort-based plan for "intersect distinct".

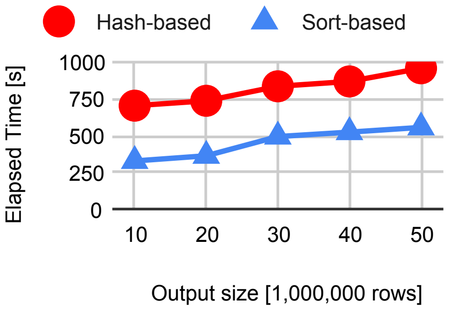

Figure 22: Cost of "intersect distinct".

6.5 A belated correction

Section 4.4 and Figure 11 of [11] compare sort- and hash-based duplicate removal, grouping, and aggregation. The overall conclusions are that sorting the input for subsequent in-stream aggregation is not competitive and that both sort- and hash-based aggregation exploit strong data reduction and small output sizes.

Figure 23 is a copy of Figure 11 of [11]. As perhaps appropriate at the time, the experimental parameters are input size $I = 100MB$, memory size $M = 100KB$, page size $P = 8KB$, merge fan-in and partitioning fan-out $F = 10$, and output size O varying from 100MB to 100KB, or from input size I to memory size M. The "group size or reduction factor" is the quotient of input and output sizes, $I \div O$. "Early aggregation" in this diagram means duplicate removal within runs on temporary storage [3]. The I/O volume reflects both writing and reading on temporary storage, i.e., the values in Figure 23 are 2× higher than the "total run size" metric used in the present paper.

Figure 24 augments Figure 23 and maybe should replace Figure 11 of [11], if that were possible. Compared to Figure 23, Figure 24 omits the curves for sorting without early aggregation and for hash aggregation without hybrid hashing, but it reproduces two of the curves using the same cost functions and parameters [11]. In addition to sorting with traditional early aggregation [3] and hash aggregation with hybrid hashing, Figure 24 shows a new curve for sort-based aggregation with both early aggregation (run generation using an in-memory index Section 3) and wide merging (a final merge step using an in-memory index Section 4). The essential observation is that the gap between sort- and hash-based aggregation, clearly visible in Figure 23, practically disappears in Figure 24. They curves are particularly close in the operating range towards the right with only a single merge or partitioning step. With todays memory sizes, most grouping operations in production workloads require only one merge

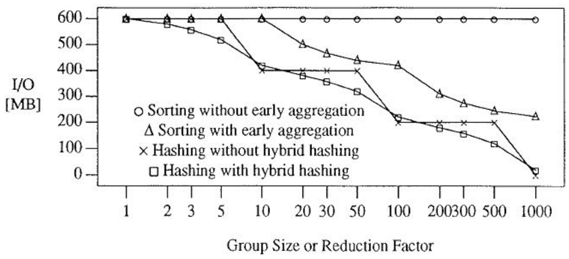

Figure 23: Prior algorithm comparison [11]

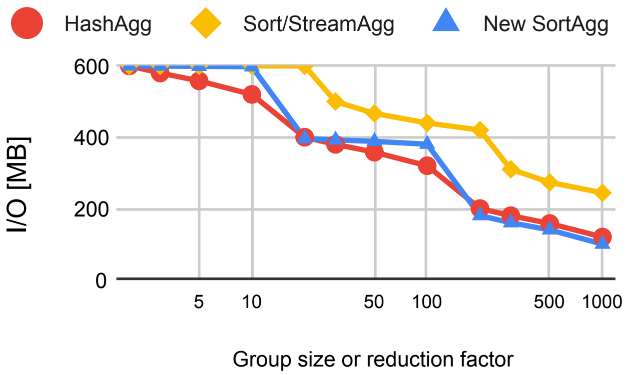

Figure 24: Revised algorithm comparison

or partitioning step. Put differently, sort- and hash-based aggregation algorithms perform very similarly for unsorted inputs. While a single diagram cannot prove it, this is true for any combination of input and output sizes. Moreover, sort-based aggregation is less susceptible to skew in the key value distribution than hash aggregation is to skew in the hash value distribution.

6.6 Summary of hypotheses and observations

In summary, our experiments confirm the calculations in our earlier examples, support our claims and hypotheses about the effectiveness of sort-based aggregation with early aggregation and wide merging, and belatedly correct an algorithm comparison published a quarter century ago.

7. SUMMARY AND CONCLUSIONS

In summary, traditional sort-based algorithms for duplicate removal, grouping, and aggregation are not quite competitive with hash-based query execution. Reflecting the current common wisdom, Mller et al. [25] state that "hashing allows for early aggregation while sorting does not." The techniques introduced in Sections 3 and 4 correct this deficiency, as shown in Section 6. For small outputs, early aggregation uses an in-memory index during run generation for read-sort-write cycles or replacement selection, spilling to temporary storage the same amount of data in the same cases as hash-based aggregation. For large outputs, wide merging uses an in-memory index during the final merge step in order to combine many more runs than a traditional merge step. These two new techniques ensure that sort-based duplicate removal, grouping, and aggregation is always competitive with hash-based query execution.

In conclusion, a single algorithm for duplicate removal, grouping, and aggregation provides multiple benefits in a query execution engine. Most obviously, it reduces the code volume and maintenance burden for query execution. Perhaps more importantly, it eliminates from query optimization the danger of mistaken algorithm choices (at least for duplicate removal, grouping, and aggregation). It also eliminates unwelcome performance surprises, unhappy users due to unpredictable algorithm choices, and engineering time wasted on analyzing execution traces. Predictable performance, in particular when combined with graceful degradation, permits smoother-running applications, more responsive dashboards, and more productive users.

8. REFERENCES

[1] Tpc-h benchmark. In *http://www.tpc.org/tpch/*.
[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107141, New York, NY, USA, 1970. Association for Computing Machinery.
[3] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Trans. Database Syst.*, 8(2):255265, June 1983.
[4] P. Boncz, T. Neumann, and O. Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*, page 6176, Berlin, Heidelberg, 2013. Springer-Verlag.
[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.
[6] W. M. Conner. Offset-value coding. In *IBM Technical Disclosure Bulletin*, 1977.
[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Googles globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug. 2013.
[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107113, Jan. 2008.
[9] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. On producing join results early. In *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '03, page 134142, New York, NY, USA, 2003. Association for Computing Machinery.
[10] R. Epstein. Techniques for processing of aggregates in relational database systems. In *Univ. of California at Berkeley, UCB/ERL Memorandum M79/8*, 1979.
[11] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[12] G. Graefe. Volcano an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120135, Feb. 1994.

[13] G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), Sept. 2006.

[14] G. Graefe. Modern b-tree techniques. *Found. Trends databases*, 3(4):203–402, Apr. 2011.

[15] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft sql server. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 8697, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[16] G. Graefe, W. Guy, and C. Sauer. Instant recovery with write-ahead logging: page repair, system restart, media restore, and system failover. *Synthesis Lectures on Data Management*, 8(2):1–113, 2016.

[17] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, page 393481, Berlin, Heidelberg, 1978. Springer-Verlag.

[18] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: A geo-replicated online data warehouse for google's advertising system. *Commun. ACM*, 59(7):117125, June 2016.

[19] T. Härder. A scan-driven sort facility for a relational database system. In *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3*, VLDB '77, page 236244. VLDB Endowment, 1977.

[20] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. SIGMOD '96, page 423434, New York, NY, USA, 1996. Association for Computing Machinery.

[21] B. R. Iyer. Hardware assisted sorting in ibm's db2 dbms. In *International Conference on Management of Data (COMAD)*, 2005.

[22] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 1625, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[23] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[24] R. P. Kooi. *The Optimization of Queries in Relational Databases.* PhD thesis, USA, 1980. AAI8109596.

[25] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-efficient aggregation: Hashing is sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 11231136, New York, NY, USA, 2015. Association for Computing Machinery.

[26] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 960971. VLDB Endowment, 2004.

[27] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[28] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, F. Weigel, D. Wilhite, J. Yang, J. Xu, J. Li, Z. Yuan, C. Chasseur, Q. Zeng, I. Rae, A. Biyani, A. Harn, Y. Xia, A. Gubichev, A. El-Helw, O. Erling, Z. Yan, M. Yang, Y. Wei, T. Do, C. Zheng, G. Graefe, S. Sardashti, A. M. Aly, D. Agrawal, A. Gupta, and S. Venkataraman. F1 query: Declarative querying at scale. *Proc. VLDB Endow.*, 11(12):1835–1848, Aug. 2018.

[29] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, page 2334, New York, NY, USA, 1979. Association for Computing Machinery.

[30] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):10681079, Aug. 2013.

[31] D. Simmen, E. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 5767, New York, NY, USA, 1996. Association for Computing Machinery.

[32] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, PDIS '91, page 6877, Washington, DC, USA, 1991. IEEE Computer Society Press.