The VLDB Journal (2023) 32:1165–1190
https://doi.org/10.1007/s00778-022-00765-x

SPECIAL ISSUE PAPER

Practical planning and execution of groupjoin and nested aggregates

Philipp Fent[1] · **Altan Birler**[1] · **Thomas Neumann**[1]

Received: 1 April 2022 / Revised: 14 September 2022 / Accepted: 25 September 2022 / Published online: 22 October 2022

Abstract
Groupjoins combine execution of a *join* and a subsequent *group-by*. They are common in analytical queries and occur in about 1/8 of the queries in TPC-H and TPC-DS. While they were originally invented to improve performance, efficient parallel execution of groupjoins can be limited by contention in many-core systems. Efficient implementations of groupjoins are highly desirable, as groupjoins are not only used to fuse group-by and join, but are also useful to efficiently execute nested aggregates. For these, the query optimizer needs to reason over the result of aggregation to optimally schedule it. Traditional systems quickly reach their limits of selectivity and cardinality estimations over computed columns and often treat group-by as an optimization barrier. In this paper, we present techniques to efficiently estimate, plan, and execute groupjoins and nested aggregates. We propose four novel techniques, *aggregate estimates* to predict the result distributions of aggregates, *parallel groupjoin execution* for scalable execution of groupjoins, *index groupjoins*, and a greedy *eager aggregation* optimization technique that introduces nested preaggregations to significantly improve execution plans. The resulting system has improved estimates, better execution plans, and a contention-free evaluation of groupjoins, which speeds up TPC-H and TPC-DS queries significantly.

Keywords Query optimization · Query processing · Parallel processing

1 Introduction

Joins and aggregations are the backbone of query engines. A common query pattern, which we observe in many benchmarks [10,59] and industry applications [77], is a join with grouped aggregation on the same key:

```
SELECT cust.id, COUNT(*), SUM(s.value)
FROM customer cust, sales s
WHERE cust.id = s.c_id
GROUP BY cust.id
```

In a traditional implementation, we answer the query by building two hash tables on the same key, one for the hash join and one for the group-by. However, we can speed up this query by reusing the join's hash table to also store the aggregate values. This combined execution of join and group-by is called a *groupjoin* [56].

The primary reason to use a groupjoin is its performance. We spend less time building hash tables, use less memory, and improve the responsiveness of this query. However, groupjoins are also more capable than regular group-bys, as we can create the groups explicitly. Consider the following nested query, with subtly different semantics:

```
SELECT cust.id, cnt, s
FROM customer cust, (
   SELECT COUNT(*) AS cnt,  SUM(s.value)
   AS s
   FROM sales s
   WHERE cust.id = s.c_id
   GROUP BY cust.id
)
```

Here, nested the query calculates a `COUNT(*)` over the inner table, which evaluates to zero when there are no join partners. Answering that query without nested-loop evaluation of the inner query is tricky, as a regular join plus group-by will produce wrong results for empty subqueries, which is known as the `COUNT` bug [58]. A groupjoin directly supports such

✉ Philipp Fent
fent@in.tum.de

Altan Birler
altan.birler@tum.de

Thomas Neumann
neumann@in.tum.de

[1] Technische Universität München, Garching, Germany

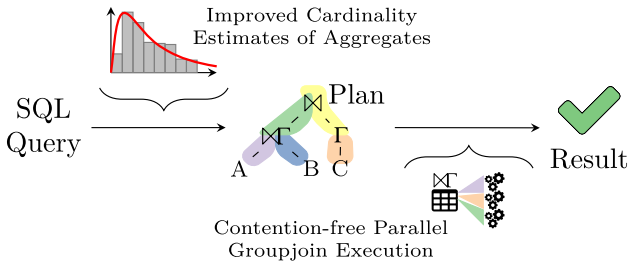


Fig. 1 Missing components for practical groupjoins. Our improvements to estimation and parallel execution enable efficient evaluation of queries with nested aggregates

queries by evaluating the static aggregate for the nested side of the join, taking the groups from the other side.

Despite their benefits, groupjoins are not widely in use. We identify two problems and propose solutions that make groupjoins more practical: First, existing algorithms for groupjoins do not scale well for parallel execution. Since groupjoin hash tables contain shared aggregation state, parallel updates of these need synchronization, and can cause heavy memory contention. Furthermore, current estimation techniques deal poorly with results of groupjoins from unnested aggregates.

The unnesting of inner aggregation subqueries is very profitable, since it eliminates nested-loops evaluation and improves the asymptotic complexity of the query. However, this causes the aggregates to be part of a bigger query tree, mangled between joins, predicates and other relational operators. Query optimization, specifically join ordering, depends on the quality of cardinality and selectivity estimates [49]. With unnested aggregates, the estimation includes group-by operations and aggregates, which are notoriously hard [28,40]. Consider the following nested aggregate with a predicate:

```
SELECT ... GROUP BY x HAVING
SUM(value) > 100
```

The result might have vastly different cardinality, depending on the selectivity, which in turn influences the optimal execution order of the query.

In our paper, we work on techniques that make combined join and aggregation more efficient, e.g., with eager aggregation [70,79] and hash table sharing via groupjoins [25,56]. In addition, we propose a novel estimation framework for computed aggregate columns, which improves the plan quality with nested aggregates. We introduce this here as part of our work in groupjoins, but the estimation framework is useful for queries with regular group-by operators, too. We integrate our work in the high-performance compiling query engine of our research database system Umbra [61]. Figure 1 shows a high-level overview of our query optimizer. On the way from an SQL query from a relational algebra query plan to the query result, we focus on efficiently evaluating nested aggregates with *computed column estimates* and *parallel groupjoin execution*.

The rest of this paper is structured as follows: First, we introduce the groupjoin and its use in general unnesting in Sect. 2. Then, we discuss and evaluate four parallel groupjoin execution strategies in Sect. 3, and propose a cost model to choose the optimal execution strategy. We evaluate the execution strategies and our cost model based on the well-known TPC-H and TPC-DS benchmarks in Sect. 4. Afterwards, we introduce our estimations for computed columns in Sect. 5 and evaluate them in Sect. 6. Furthermore, in Sects. 7 and 8, we improve query plans by considering groupjoins for operator ordering and propose an eager aggregation strategy. Section 9 discusses the impact of our work on queries from TPC-H, before we discuss related work in Sect. 10, and conclude in Sect. 11.

2 Groupjoin for nested aggregates

Apart from better performance, the semantics of groupjoins are useful to compute nested aggregates. Due to the versatile subqueries in SQL, aggregates can appear in various places of the query plan. To efficiently calculate such aggregates, it is important to unnest and not evaluate them in nested-loops [6,34,62]. However, decorrelated aggregates need a careful implementation and are challenging for query planning.

2.1 Groupjoin

We define a groupjoin $\bowtie\!\!\!\!\Gamma$ [56] as an equi-join with separate aggregates over its binary inputs grouped by the join key.

$$
\begin{aligned}
&R \bowtie\!\!\!\!\Gamma_{a_1=a_2:agg} S := \{r \circ [g_r : G_R] \circ [g_s : G_S] \mid r \in R,\\
&G_S = agg(\{s \mid s \in S \wedge r.a_1 = s.a_2\}),\\
&G_R = agg(\{r \mid r \in R \wedge r.a_1 = s.a_2\})\}
\end{aligned}
$$

We further require that $a_1 \rightarrow R$, i.e., that the join condition functionally determines R. With this definition, we compute an equi-join between R and S on a key of R, and compute aggregates separately over the matching tuples, which can be beneficial since we can avoid duplicate tuples of R and building a duplicate hash table.

The intuitive use-case for groupjoins is an optimization to fuse a join and a group-by operator, given that the preconditions shown in Fig. 2 are satisfied, and we can separately evaluate the aggregates: ① The join and aggregation keys need to be equivalent, and ② these keys are superkeys w.r.t. functional dependencies of the left build side. In this case, introducing a groupjoin is usually considered to be a net

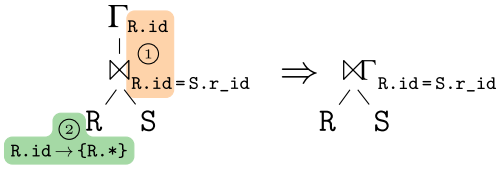


Fig. 2 Preconditions to introduce a groupjoin

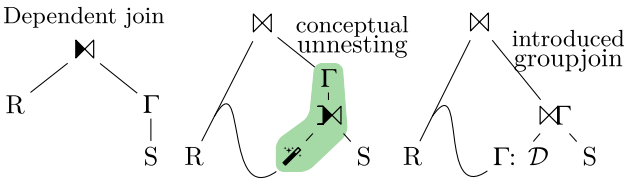


Fig. 3 General unnesting: decorrelation of dependent subqueries containing an aggregation can introduce a groupjoin

win [16,25] and can reduce the cost of those operators by up to 50 % by eliminating intermediate results.

2.2 Correlated subquery unnesting

The groupjoin also supports the challenging edge cases of whole table aggregates in a correlated subquery. Consider the correlated subquery from Sect. 1, where we calculate a whole-table `COUNT(*)` on sales that is correlated with the outer query's customer. Conceptually, we need to calculate a whole table aggregate for each customer, but ideally want to introduce a more efficient join. However, using an outer join is tricky, since we cannot directly translate whole table aggregates to the join result. A groupjoin can instead evaluate the left and right sides separately, where a careful initialization can produce equivalent results to whole table aggregates. For the `COUNT(*)` example, we initialize empty groups (e.g., customers with no sales) as zero, and increment it with whole-table tuple counting logic[1].

For the general case, we deliberately introduce a groupjoin to separately calculate the aggregates of the correlated subquery, filter unnecessary tuples, and avoid the `COUNT` bug [62]. Figure 3 shows this unnesting for two arbitrary tables R and S, with the dependent subquery-join ⋈ on top and a nested whole table aggregate Γ in the correlated right subtree. To decorrelate this aggregate, we first compute the magic set 🪄 of relevant tuples for the correlated subquery [73]. To compute the set, we eliminate any duplicates of the outer-side join key with a group-by Γ and get the precise domain $\mathcal{D}$ of potentially equivalent keys for which we need to calculate the inner aggregate. With this condensed set of outer keys, we satisfy both preconditions to introduce a groupjoin,

[1] `COUNT(*)` has some edge cases that are trivial in a groupjoin, but difficult in separate operators. See Sect. 3.3 for an extended discussion.

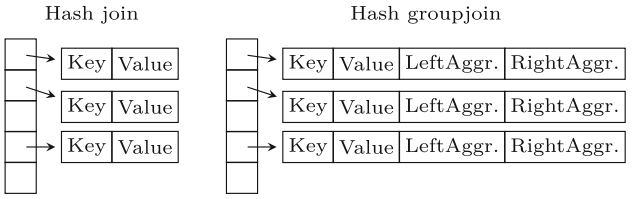


Fig. 4 Single threaded groupjoin Hash table. Aggregates from either join side are materialized as hash table payload

which we use to keep the aggregation of the subquery side S separate.

In the following, we parallelize groupjoins with on-the-fly adaptive data segmentation into morsels and contention-avoiding relational operators that allow dynamic work-stealing.

3 Parallel execution of groupjoins

The parallel execution of common relational operators is widely studied and efficient parallel join and aggregation algorithms are used in many systems that can scale analytical workloads [14,38,63]. Groupjoins, which fuse join with aggregation hash tables, promise a significant speedup in comparison to separate operators and are necessary for general unnesting. However, parallel execution of groupjoins can be a bottleneck due to contention. While several publications have previously discussed groupjoins, they are now well over a decade old and single-threaded [12,15,53].

Figure 4 shows a basic, single-threaded implementation of a groupjoin, and its similarity to a regular hash join. In this example, we use a hash table to store the hash table payload, which includes the accumulators for the aggregates of both sides. During the build phase, we initialize these as empty to support the semantics of static (whole table) aggregation.

In contrast to joins, the probe phase of groupjoins is not read-only, but needs synchronization of the aggregate updates when using more than one thread. The shared state of the aggregates poses a problem for parallel execution, and we need synchronization, e.g., with fine-grained locking, to avoid data races. Unsurprisingly, the synchronization overhead can quickly become a bottleneck, especially in the presence of heavy-hitters [67]. While updating the aggregates is generally a quick operation, and the critical section only spans a couple of instructions, all threads will compete for the same locks of the heavy-hitters. Even when eliding this lock and updating the aggregates with lock-free atomic instructions, memory contention, which is the root-cause for this bottleneck, still remains a problem and causes suboptimal performance.

In the following, we propose three execution strategies for groupjoins that avoid synchronization between threads.

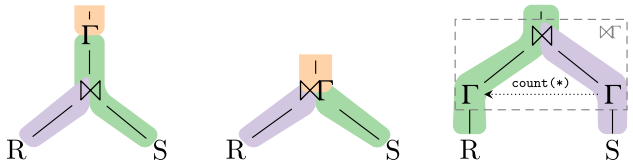


Fig. 5 Eager grouping. While the middle groupjoin eliminates the second hash table, the schematic eager aggregation on the right can additionally eliminate the result scan

For each implementation, we discuss, in which scenarios it is an efficient implementation of a groupjoin. Based on these insights, we propose a cost-based strategy in Sect. 3.5, to choose the best physical plan, depending on the underlying data distribution.

3.1 Eager right groupjoin

One well-known technique of aggregation queries is eager aggregation [79]. A group-by can be pushed down, past a join, to reduce the number of input tuples to the join. In the general case, this needs an additional group-by after the join, since the join might have a multiplying effect on the aggregate tuples. In this section, we apply eager aggregation to groupjoins: When we can speculate that almost every tuple finds a join partner, i.e., the relative-right selectivity σ_S is close to 1, then eager aggregation will substantially reduce the number of tuples that need to be processed by the join.

When eagerly aggregating in a groupjoin, we can exploit several facts that allow making eager aggregation very efficient: Precondition ② (cf. Sect. 2.1) of the groupjoin guarantees that the join and group key of the left-hand side functionally determine the left tuples. In other words, the left side does not contain duplicates and, thus, cannot have a multiplying effect on the right aggregation. As sketched in Fig. 5, we can exploit this fact by first eagerly executing the right aggregation. If there are any aggregates on attributes of R, duplicate keys in S have a multiplying effect that duplicates the keys but do not change their value. We account for this effect with a `count(*)` aggregate on S, which we apply as multiplication factor of the unique tuples of R. In result, we elide the final group-by that would be needed for general eager aggregation as described by Yan and Larson [79], and replace the result scan with a single hash table probe.

We eliminate the result scan, and improve the pipeline behavior, by using the same precondition ②. A group-by is a *full pipeline breaker* [60], i.e., it materializes all incoming tuples and scans the result when the last tuple was processed. However, this flushes all data from CPU registers, or very hot cache, which makes pipeline breakers expensive. Algorithm 1 shows pseudocode to execute this operator, where each loop represents one pipeline. In the first loop, we eagerly aggregate the whole right side S into the aggregation hash table Γ_S. The second loop probes with the left side R and calculates the complete left aggregate in a single step with the probe result. Afterwards, the loop still is not terminated, but can continue its pipeline with any next operations, in this case `output`.

Algorithm 1 Example code generated to execute an eager right groupjoin with inner-join semantics.

```
initialize memory of Γ_s
for each tuple s in S
   aggregate s as a_s in hash table of Γ_s
for each tuple r in R
   if r has match a_s in Γ_s # inner
   groupjoin
      a_r := agg(r * a_s.count(*))
      output: r ∘ a_r ∘ a_s
```

In contrast to a lazily aggregated groupjoin, eager aggregation requires no explicit synchronization through locks. Our implementation reuses the implementation of regular aggregation, which first builds partitioned, thread-local aggregation hash tables [42,46,69]. A second step exchanges these partitions between threads and merges them into a partitioned global result hash table. Afterwards, the duplicate-free left side can exclusively read its matches in the hash table, which allows contention-free and full parallel execution.

While it can be executed very efficiently, eager aggregation is no one-size-fits-all solution. Depending on the relative right selectivity of the join part, i.e., how many groups of the right-hand side are not matched by the left, we might calculate many unneeded aggregates. Therefore, we deem it necessary to only use this eager aggregation, when a local cost-model predicts it to be beneficial.

The following cost function models the eager right groupjoin and closely follows the presented algorithm:

$$C_{\text{eager}} = |S| + |R \ltimes S|$$

First, we build the eager hash aggregation in two passes over the data, which touches every tuple of S twice: $2\,|S|$. Then, we probe the hash table with the left-hand side $|R|$, and check the matching tuples $|R \ltimes S|$, for equality. In our cost function C_{eager}, we exclude the initial passes over each input side $|R| + |S|$, which are required by any groupjoin implementation. Nevertheless, we include the result scanning phase of pipeline breakers, to differentiate operator-fused pipelines that do not need to materialize their result.

3.2 Memoizing groupjoin

Eagerly aggregating is very beneficial, when almost every right tuple finds a join partner. The other extreme is also com-

mon, i.e., that many tuples are filtered by the join. In this case, we want to filter right-hand side tuples, before aggregating them. In the following, we present a groupjoin implementation that builds filtered thread-local aggregates and efficiently merges them to a groupjoin result.

The idea of this implementation is to optimistically use a shared global aggregation hash table for aggregates with few tuples, but aggregate heavy-hitters thread-locally. The global hash table resembles the sketched single-threaded groupjoin in Fig. 4, where we first build a join hash table with the left-hand side R with additional space for the aggregates. For synchronization, we use an atomic set-on-first-use thread-id tag that assigns groups to the first thread that updates it. Additionally, when we probe the hash table with S, we memoize the payload pointer to avoid a duplicate lookup.

Algorithm 2 Memoizing groupjoin probe pipeline with ownership tagging.

```
1  Hashtable globalHt
2  // Omitted: Concurrent build of R hash
3     table
4  thread_local localHt, tid
5  for each tuple s in S
6    hash := hash(s.key)
7    *p := globalHt.probe(hash, s.key)
8    if p not found
9      continue
10   owner := p->tid.atomic_load(relaxed)
11   inPlace := owner == tid;
12   // Is uninitialized?
13   if owner == 0
14     inPlace = p->tid.CAS(owner, tid)
15   if inPlace
16     p->aggregate(s)
17   else
18     localHt[hash, p].aggregate(s)
```

The intent behind this hybrid synchronization strategy is to avoid tiny thread-local groups with very few tuples, while still aggregating heavy-hitters thread-locally. With the thread-id tags, singleton groups, and groups that are clustered on a single thread, directly use the result hash table, which reduces the size of local hash tables that would later need to be merged again. In effect, this reduces the partial aggregates to the number of threads n, compared to $n + 1$ for full thread-local preaggregation and merging into a global hash table.

Algorithm 2 shows pseudocode for the described groupjoin probe pipeline. The atomic operations here use a memory model akin to the C++ model [8]. For our optimistic synchronization, we use a single atomic compare-and-swap (CAS), which is the only operation that requires memory synchronization. The low-cost relaxed read of the current tag in line 9 does not need synchronization and could read stale data. For correctness, in the sense of being free of data races, this read is not required. However, it is a vital optimization for heavy-hitters, where the CAS synchronization would cause memory contention. Instead, after the initial CAS, any heavy-hitters will not take this branch again and all other operations are either non-atomic or relaxed. In result, this thread-local preaggregation is virtually contention free. Afterwards, when all input data was either aggregated locally or globally, we exchange the local partitions between threads.

In the thread-local aggregation, we reuse previously calculated intermediates. The local hash table lookup reuses the hash of the global hash table lookup, and, instead of comparing the full key for equality, we only check if the pointer of the probe result from line 6 matches. We also store just this pointer in the local tables, which we also use as a shortcut for merging the aggregates. When all probes from R are finished, we merge the thread-local groups by following this memoized probe pointer, which reduces the number of cache misses and avoids a second hash table lookup.

Compared to the eager right groupjoin, this memoizing approach favors small left sides with a selective join. Expressed more formally for our cost model, we use a build of the left hash table in two passes $2\,|R|$, probe once with the entire right side $|S|$, before checking the matching tuples $|R \rtimes S|$ for equality. Then, we use these to build thread-local aggregates, before merging them into their memoized global bucket, $2\,|R \rtimes S|$. Since this variant of the groupjoin is a full pipeline breaker, we additionally need to scan the entire $|R|$ hash table to start the next pipeline, while omitting unjoined results. In sum, we arrive at the following cost function:

$$C_{\text{memo}} = 2\,|R| + 3\,|R \rtimes S|$$

3.3 Separating join and group-by

As laid out in Sect. 2, groupjoin has its own semantics that is useful for whole-table aggregates of unnested queries. An alternative to a dedicated operator would be to emulate this behavior with reused join and group-by operators, which reduces the implementation overhead, but might build duplicate state in two hash tables.

This duplicate state was the reason that previous work [16,25,56] considered a groupjoin as unconditionally advantageous to a separate execution. However, a careful analysis of the involved operations shows that there exist cases where a groupjoin is more expensive than a separate inner join followed by a group-by. The intuition behind this somewhat counter-intuitive finding is that the groupjoin result set might be bigger than that of a separate group-by. That is, when the join is selective on the left build side R, then the join-

Table 1 Static count semantics

R ⟕ S	`count(*)` subquery	`count(S)` after ⟕	`count(*)` before ⟕
(NULL, NULL)	1	**0**	1
(1, 1)	1	1	1
(2, NULL)	0	0	**NULL**

In separate operators `count(*)` aggregates might produce different results
Bold values are different results

reduced aggregate table will be significantly smaller than the join table. In this case, it is cheaper to probe a separate join table and build a densely populated aggregation table instead of reusing the relatively sparse matches in the join table. In the following, we show how a groupjoin can be rewritten as a join and group-by, while still preserving the static aggregation semantics to unnest arbitrary queries (cf. Sect. 2.2).

While for most groupjoins, the separation into a join and group-by is trivial, the ungrouped whole table aggregations that can appear in correlated subqueries require special care to preserve their semantics, especially with `NULL` values [15, 74]. We call this special case a *static groupjoin*. Consider the following example of a query that we process with such a static groupjoin:

```
SELECT r.id, cnt FROM R r, (
  SELECT COUNT(*) cnt
  FROM S s
  WHERE r.id IS s.r_id)
```

Our general unnesting resolves the correlated subquery with a groupjoin. The following shows the resulting plan in SQL-like syntax:

```
SELECT r.id, COUNT(S::*) FROM R r
STATIC LEFT GROUP JOIN S s
ON r.id IS s.r_id
```

The important distinction of the static groupjoin is between empty inner tables and `NULL` values. Table 1 shows three cases, where the aggregated count differs: A `count(*)` in a subquery counts any matching tuple, even when its value is `NULL`. Executing an outer join R ⟕ S, produces additional `NULL` values that need to be ignored by a count of S tuples. However, with a separate aggregation operator, a naïve count cannot distinguish between matches, where `NULL IS NULL` and padded tuples that did not have a join partner. Even evaluating the aggregates before the join would still require coalescing of `NULL` aggregates. To execute the join before aggregating, we ensure the correctness of the aggregates with a join marker that decides between ignored and `NULL` tuples:

```
SELECT r.id, COUNT(s.joinMarker)
FROM R r LEFT OUTER JOIN (
  SELECT *, TRUE AS joinMarker FROM S
) ON r.id = s.r_id
GROUP BY r.id
```

Rewriting such a groupjoin as `LEFT JOIN` is usually not beneficial for performance, since it fixes the relative left selectivity to one. On the other hand, most groupjoins do not need an outer join, and might be cheaper executed in separate hash tables. For our cost model calculation, we first two-pass build a $2\,|R|$ hash table, then probe with $|S|$ and match $|R \rtimes S|$ right tuples. With the resulting tuples, we build a separate aggregation table, again in two passes $2\,|R \rtimes S|$, before we scan the $|R \ltimes S|$ matched aggregation groups. The drawback in comparison to the memoizing approach is that we do not know the size of the aggregation state beforehand. Therefore, we need to additionally check if the aggregate already exists, and dynamically allocate and initialize memory on demand. While this can reduce resource usage for unmatched keys in R, the fine-grained allocations are more expensive per match ($|R \rtimes S|$) than a bulk operation for all keys. In our simplified cost model, we express this as a fixed factor, which we measured empirically as $c = 30\%$ overhead. In total, we arrive at the following cost function:

$$C_{\text{sep}} = |R| + (3 + c)\,|R \rtimes S| + |R \ltimes S|$$

3.4 Using indexes for groupjoins

The previous execution strategies are designed to work over arbitrary inputs. That means, we always need to build a data structure to aggregate values during execution. For join processing, one can often use indexes to access a base table relation on one side of the join [72]. Using indexes to support aggregations can also be beneficial when applied properly.

Some DBMSs already use index scans to filter and compute group-by aggregates pipelined. Similarly, we can avoid building a separate aggregation data structure and use the index for a groupjoin implementation that aggregate the group with little overhead. The idea here is similar to the eager right groupjoins (cf. Sect. 3.1), where we probe the right side aggregation hash table. However, for already existing indexes, we do not have matching aggregates, but need to calculate them during the index probe. We can do this efficiently, since the left side is duplicate free, and we visit all elements of the right group during a regular index probe.

Using indexes is especially fitting for groupjoins, since we operate on a key of the left side. When we have a key and join with a base relation, this usually means that there exists a foreign key constraint with a corresponding index. Thus, we likely already maintain matching indexes for groupjoins and can use them for a more efficient execution.

Using already existing indexes for groupjoins has several advantages: By using the index to find matching join partners, we avoid accessing unrelated tuples, e.g., when the relative right selectivity σ_s is very low. Also, we need a minimal amount of working memory, since we only keep the aggregation state for the current index probe. As a consequence, e.g., when we calculate a single sum, we can keep the aggregate in a dedicated CPU register and get excellent performance.

Algorithm 3 Example code generated to execute an index groupjoin with inner-join semantics.

```
for each tuple r in R
   a_s ≔ ∅
   for each matching tuple s in S.index
      :aggregate s in a_s
   if at least one match: # inner
   groupjoin
      a_r ≔ agg(r * a_s.count(*))
      output: r ∘ a_r ∘ a_s
```

Algorithm 3 shows pseudocode to execute such an index groupjoin. The outer loop represents the input pipeline, in this case a table scan, but we can also operate on arbitrary input tuples, e.g., from a join result. For each input tuple, we initialize an empty aggregate a_s, before we probe the index for matching tuples and combine them in this local aggregate. After probing the index, we report the result to the next operator, depending on if we had at least one join partner, or if we need to report static (cf. Sect. 3.3) results.

In contrast to the methods presented in the previous subsections, probing an index has some inherent limitations in parallel execution. While we can probe the index in parallel with multiple threads, scanning the matching tuples in the index is harder to parallelize. This is especially problematic for heavy hitters, e.g., when a single left tuple finds millions of join partners on the right. Then, it is unattractive to aggregate this heavy hitter single-threaded.

In this case, it is advantageous to split the equality ranges into schedulable morsels and use multiple threads to aggregate in thread-local storage. This, however, looses the pipelining benefits of the index-based groupjoin operation and effectively executes the groupjoin separately. Defending against this case requires metadata in the index to detect the presence of such heavy hitters. When this is the case, we fall back to a separate execution as presented in Sect. 3.3.

The JCC-H [9] benchmark demonstrates these problems, where there are five populous orders that have very many lineitems. When we groupjoin these orders with the lineitem table, the index based execution is essentially single-threaded, while the memoizing and eager right execution execute on all available cores. In this scenario, the optimal method additionally depends on the available cores of the machine.

The cost calculation for this execution strategy differs significantly from the other strategies. Since we choose a different access path for the tuples of the right side S, the decision to use an index is strongly dependent on the relative performance of accessing data linearly during a table scan, or using the random-accesses of an index. This performance depends on many factors: What kind of index do we use? Traditional B-Trees [4,30], or in-memory optimized indexes such as lock free hash indexes [20] or Adaptive Radix Trees [48,50]. Additionally, the random-access performance also depends on the physical storage of the base table. For example, cloud-centric storage architectures using large files [17] have large read amplification for random lookups, and even storing tuples in main-memory optimized compressed Data Blocks [44] introduces some overhead.

We, therefore, exclude index groupjoins from our regular cost models. In our system Umbra, we instead use a two-stage optimization, where we first decide if we use the index, and if not choose one of the other execution strategies. In Umbra, we use an empirically determined $10\times$ overhead of accessing tuples via an index join with a B-Tree index and cached pages, compared to a full table scan.

When we have a suitable index and accessing it is cheaper than the full table scan, index groupjoins have excellent performance. In comparison to the other combined methods, we can avoid scanning the full right table. In separate join and group-by execution, we also use the index, but still break the execution pipeline by building the aggregation hash table. By avoiding this unnecessary hash table, index groupjoins are about 33 % faster than separate index joins.

3.5 Choosing a physical implementation

To recap, we presented four parallel execution strategies for groupjoins. In Sect. 3.1, we presented an eagerly right aggregating groupjoin, in Sect. 3.2 we used a combined join and aggregation table with memoizing thread-local aggregations. Furthermore, we showed in Sect. 3.3, that we can rewrite arbitrary groupjoins as separate join and group-by. Lastly, we described how to use indexes for groupjoins in Sect. 3.4. All four implementations have different characteristics, which we formalized for the first three as a cost model to compare

Table 2 Example cost calculations of groupjoin implementations

$\|R\|$	$\|S\|$	σ_R (%)	σ_S (%)	$\|R \ltimes S\|$	$\|R \rtimes S\|$	C_{eager}	C_{memo}	C_{sep}
100	200	80	80	80	160	**280**	680	708
100	200	80	10	80	20	280	260	**246**
100	100	100	10	100	10	**200**	230	233
100	500	100	5	100	25	600	**275**	283

Bold values mark the best value

their relative performance:

$$C_{\text{eager}} = |S| + |R \ltimes S|$$
$$C_{\text{memo}} = 2\,|R| + 3|R \rtimes S|$$
$$C_{\text{sep}} = |R| + 3.3\,|R \rtimes S| + |R \ltimes S|$$

The base of all three cost functions consists of the underlying cardinalities $|R|$ and $|S|$, and the semijoin reduced cardinalities $|R \rtimes S| = |R|\,\sigma_R$ and $|R \rtimes S| = |S|\,\sigma_S$. In Table 2 we go through some exemplary calculations of this cost model. As the examples show, the different implementations have significant differences in the total cost of execution, depending on how much a side is reduced with its relative selectivity σ.

When considering C_{eager}, the differences are especially pronounced. C_{memo} and C_{sep} are closer, since both approaches implement similar logic. Their largest difference is the static vs. dynamic memory allocation to compute the aggregates. Figure 6 shows the allocated memory in Umbra during the execution of a groupjoin with our three implementations. In the shown case, every input tuple finds a join partner, thus we need memory to store all tuples. Both fused approaches store them in one hash table, either statically allocated up front (memoizing), or dynamically during eager aggregation of S. In contrast, separate execution allocates a smaller initial hash table and dynamically builds the additional aggregation table. In this example, the fused storage uses about 1GB peak memory, while separate execution consumes about 50% more. However, depending on how many distinct aggregates we encounter (σ_R), the dynamic allocation of the separate execution might also use less memory. In our cost model, we encode this difference as the simplified 30% factor in C_{sep}. However, this factor depends on a few system characteristics, e.g., the cost to dynamically allocate memory and the momentary scarcity of it. Additionally, the number of aggregates also influences the hash table payload sizes.

Like any cost-based optimization, this approach relies on estimates of the underlying data. While this works well for base tables and joins, the quality can deteriorate with nested groupjoins and other aggregates.

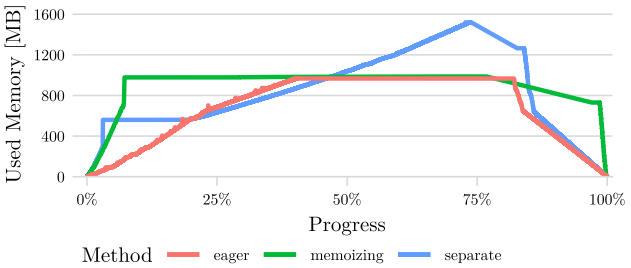


Fig. 6 Memory consumption of TPC-H SF 10 `orders` - `lineitem` groupjoins

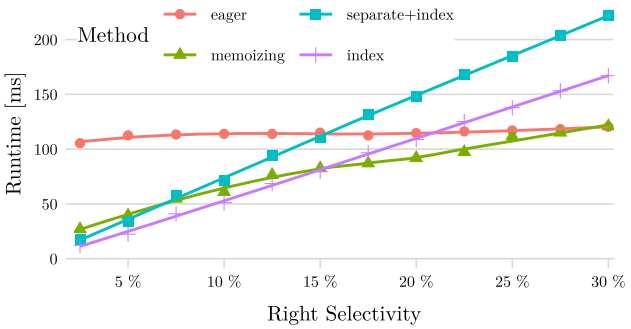


Fig. 7 Performance of the index groupjoin on TPC-H SF 10 `orders` - `lineitem` via the `l_orderkey` foreign key

4 Evaluation of groupjoins

In this chapter, we present the experimental evaluation of the presented groupjoins in our research RDBMS Umbra [61]. We start with a study of the behavior of parallel groupjoin execution in the TPC-H benchmark, and if it corresponds to our presented cost model. As detailed in Sect. 3, groupjoins are commonly used in unnesting, but we also apply them when they can improve performance. For this evaluation, we consider the groupjoins in the well-known analytical benchmark TPC-H, compare the performance of our proposed implementations, and evaluate our cost model therein.

Hypothesis For TPC-H, the selectivity and relative sizes do not change when increasing the scale factor, thus our cost model presented should stay consistent relative to each variant. Since all three proposed algorithms are virtually lock

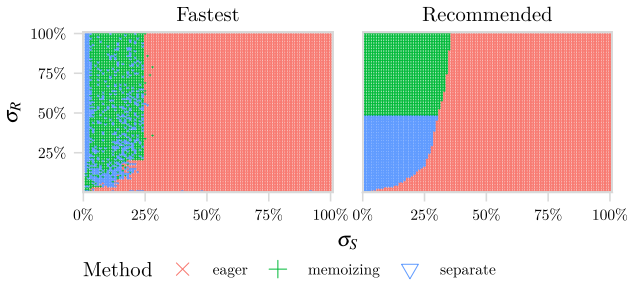


Fig. 8 Comparison of fastest and cost model recommended implementation for a TPC-H `orders - lineitem` groupjoin

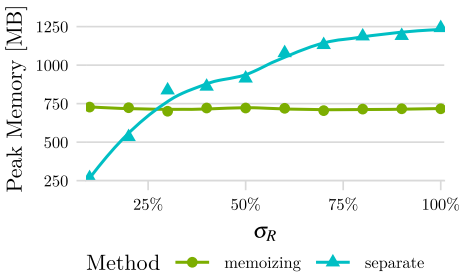


Fig. 9 Peak memory usage for a TPC-H `orders - lineitem` groupjoin

and contention free, we expect no relative changes between algorithms under varying parallelism or data size.

Setup of performance measurements We run all benchmarks on a NUMA system with 2× Intel Xeon E5-2660v2 CPU with 10 cores each, 2× hyper-threads, and 256GB RAM. To measure performance with warm caches, we repeat the executions 20 times and report the median value. The typical run-to-run median absolute deviation for this setup is 1%.

In a first experiment, we evaluate the performance of our different groupjoin implementations under a varying right selectivity σ_s. Figure 7 shows the execution time of our different groupjoin implementations. In this experiment, we groupjoin the TPC-H orders and lineitem tables with a foreign key index on `l_orderkey`, which we use either in an index join or an index groupjoin. In a direct comparison, index groupjoins are strictly better than building a separate aggregation hash table when σ_s is small. However unsurprisingly, when we join with more tuples of lineitem, the memoizing and eager right approaches can be faster. Index groupjoins are faster for a right selectivity of up to $\sigma_s = 15\%$. Compared to this, Umbra's 10× heuristic is rather conservative.

In the second experiment, we validate the quality of our cost model recommendations. This experiment compares the predicted cheapest to the actually measured fastest implementation. The setup is a micro benchmark on the TPC-H SF 10 data set with the same, single `orders-lineitem` groupjoin. To test the whole σ_R and σ_S parameter space, we prefilter each of the tables in 1% increments via the primary key. Figure 8 shows the 10,000 combinations, and plots the measured fastest implementations in the left plot, in comparison to the cost model recommended ones on the right.

This experiment shows that our prediction is a good indicator of the actual fastest performance. As expected from the cost model, the most impactful decision is whether we should aggregate eagerly. Our cost model recommends this for the upper two-thirds of the σ_S range, while the measurements indicate that the break-even point is already a bit lower. However, at this border the methods only have minor performance differences. To quantify this, we pairwise compare the performance of the measured fastest method with the, sometimes slower, cost model recommendation. Using the recommendations results in a mean absolute percentage error of only 1.7% over the best performance and a maximum absolute percentage error of 95%.

However, the memoizing and separate execution strategies are generally closer in their measured runtime performance. We attribute this mostly to the optimized dynamic memory allocation in Umbra [22], since the peak memory usage differs much more. To quantify this, we measure the maximum amount of memory used to execute the groupjoin under a varying left selectivity σ_S in Fig. 9.

In the next experiment, we limit the amount of parallelism and observe the query performance with a fixed groupjoin algorithm, and fixed TPC-H scale factor 10. The third experiment uses all threads, but varies the scale factor. Note that Umbra was already a system with state-of-the-art performance, even without our contributions. As baseline for TPC-H, the speedup of Umbra over MonetDB [11] is about 3.2× and about 101× over PostgreSQL [76].

Cost model We first go through the cost model calculations for groupjoins in TPC-H, before evaluating if the model accurately predicts the performance of these queries. For this evaluation, we look at a total of four TPC-H queries using a groupjoin: Q3 is an organically occurring groupjoin, where we first join and then group-by the same key. Q13 has a similar groupjoin sequence, albeit in a nested query itself. In contrast, the groupjoins in Q17 and Q18 are the result of unnesting. We also provide an interactive query plan viewer for these queries online.[2]

The cost model calculations for these joins in Table 3 show our predicted relative performance for these queries. Q3 has high selectivity of the right-hand side, which favors the lazily aggregating variants, and a moderate relative left selectivity, which puts separate processing at an advantage. When we look at Q13, the join is very unselective on the right side,

[2] https://umbra-db.com/interface/.

Table 3 TPC-H Groupjoins. Cost model calculations with four TPC-H groupjoin queries on scale factor 1

Q	$\|R\|$	$\|S\|$	σ_R (%)	σ_S (%)	C_{eager}	C_{memo}	C_{sep}
3	147k	3.24M	54	6.8	3.32M	956k	**954k**
13	150k	1.48M	63	100	**1.58M**	4.75M	5.14M
17	204	6.00M	100	0.10	6.00M	**19.0k**	20.9k
18	57	6.00M	100	0.84	6.00M	**152k**	167k

Bold values mark the best value

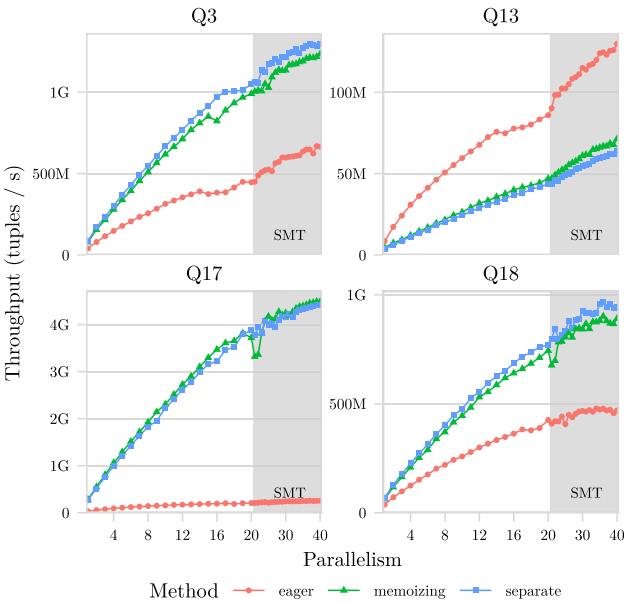


Fig. 10 Parallel scale-out of TPC-H SF10 groupjoin queries

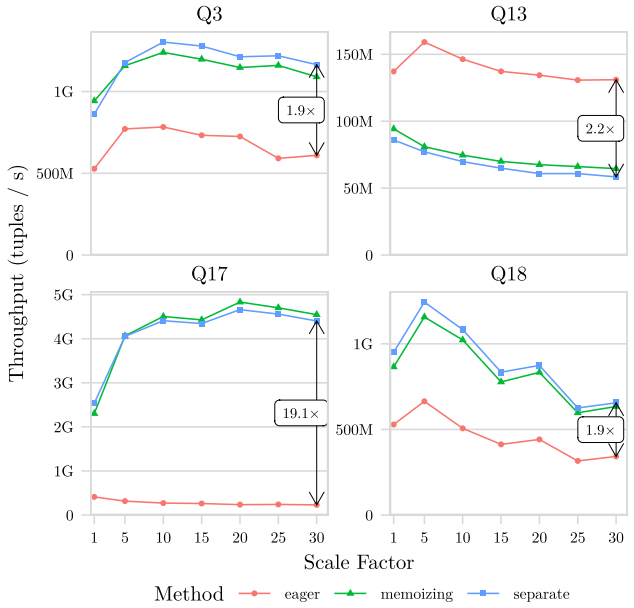


Fig. 11 Data size scale-out of groupjoins in TPC-H

which puts eager right aggregation at a clear advantage. Both unnested queries Q17 and Q18 only compute the groupjoin on a small and highly selective left side, which puts the hybrid memoizing groupjoin at a slight advantage.

In the following, we run two experiments of our algorithms under a varying parallelism and data scale to validate these claims and to show that the cost model calculations are robust under these parameters. In contrast to the cost calculations from Table 3, which only include the variable costs of the groupjoin implementation, our benchmarks measure the throughput of the whole query.

Figure 10 shows the relative performance of the different groupjoin implementations with increasing parallelism. We observe that, as expected, the relative performance between the algorithms stays the same. All three implementations show a linear speedup when increasing the parallelism, with a tamping down speedup on hyper-threads.

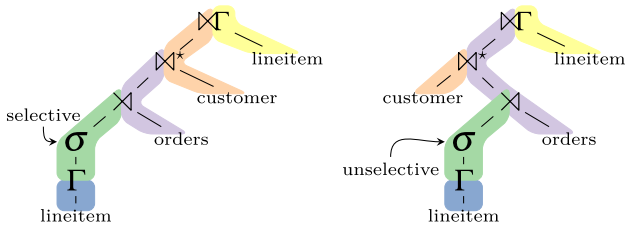
In Fig. 11, we vary the amount of processed data via the scale factor and see a similar picture. Again, the relative performance stays unchanged and, apart from some effects when exceeding cache sizes, the overall throughput stays relatively constant. All in all, our cost model has proven to be robust in regard to variable system parameters, and accurately predicts the most efficient groupjoin implementation.

Overall, eager aggregation can bring over 2× improvement in Q13, but is over an order of magnitude slower in Q17. The other implementations are much closer to one another, mostly because we build the hash table with the duplicate free left side, which is orders of magnitude smaller than the right side. In comparison to processing the large right side, building the relatively small left hash table has only a minuscule impact on the overall query. Nevertheless, a proper model will find the best execution plan and significantly improve the efficiency.

Over the four queries in TPC-H that use a groupjoin, our cost model based approach achieves a geometric mean speedup of 20% over a baseline that executes join and group-by separately. We also ran a similar experiment over TPC-DS, where we see similar results: A total of 13 queries can use a groupjoin, with a geometric mean speedup of 5%.

5 Aggregate estimates

Good estimates for computed columns in nested aggregates are one of the missing links in cost-based query optimization. Cardinality and selectivity estimations for base table columns are well-known, and despite some problems, work quite well in practice [47,63]. While statistics on singular columns fail to capture correlations, histograms, samples, and sketches provide a solid baseline, and recently developed techniques using machine-learning work towards multi-column estimates [23,41]. However, estimates for computed columns



(a) Selective σ-Predicate **(b)** Unselective σ-Predicate

Fig. 12 Possible query plans for TPC-H Query 18. Depending on the σ filter selectivity, we use the customer relation as hash-join build or probe side, which roughly leads to a 10% difference in performance

such as aggregates are rarely used, which results in poor cost-model calculations, and suboptimal query plans.

We propose to extend existing approaches that work on base table columns by calculating statistics, which allow deducing computed column estimates. Our approach uses a lightweight statistical model that can be piggybacked onto regular sampling or histogram-based statistics. The key idea is to fit a skew-normal distribution to the underlying data using a method of moments estimator, which can be cheaply maintained on base tables, as well as for computations throughout the query tree. With this fitted distribution, we then efficiently estimate the selectivity of predicates on computed columns, and the resulting cardinality.

Surprisingly few systems consider the results of computed columns in cardinality estimation, which is rather surprising considering this is a part of standard SQL, which even has a dedicated `HAVING` syntax. After unnesting or in nested analytical views, it is common to have aggregates and predicates on aggregates embedded in lower parts of the algebra tree, where the resulting cardinality has consequences for the quality of query plans. One example is TPC-H Q18 shown in Fig. 12, with the nested predicate `HAVING SUM(l_quantity) > 300`. The estimated selectivity for the filter σ in the green pipeline has significant impact on the query performance. Depending on the selectivity, the optimal query plan is either, a) when the predicate is very selective, or b) if it is not.

In the figure, we use the convention to build the join hash tables with the left and probe with the right side. For Q18, all data sources except the aggregate result are unfiltered base tables, where cardinality estimation is trivial. The challenging part for cardinality estimation is the join with the customer table $\bowtie^\star$, which is marked with an asterisk. Since building a hash table is more expensive than probing it, we estimate which side is smaller. In Q18, we estimate if the σ filter condition produces less tuples than the entire customer table. The base table cardinalities differ by an order of magnitude (150k customers and 1.6M distinct orders for scale factor 1), so simple heuristics most likely mispredict these cardinalities. In preliminary experiments, this misprediction has roughly a 10% performance penalty for the whole query. To avoid this and get closer to the real selectivity of .003%, we need robust estimation of computed columns.

In the following, we present our novel computed column estimator, based on the method of moments for skew-normal distributions [66]. In result, we get orders of magnitude better estimates for filters on computed columns and in turn generate better query plans.

5.1 Skew normal distribution

Our key insight is that `HAVING` predicates are mostly on computed values based on columns of "natural" numerical quantities, e.g., price, balance, counts, ratings, durations, etc. In contrast to predicates on keys or identifiers, they are rarely compared for equality, but more commonly with range predicates, e.g., $\leq$ or `BETWEEN`. In the following, we propose an estimation model for computed columns that roughly follow a normal distribution, i.e., most values center around a mean, with relatively few outliers from that mean. Additionally, we model a limited amount of skewness in the underlying data to break the inherent symmetry of a pure Gaussian normal distribution. The resulting selectivity estimation framework then handles a wide variety of computed columns.

The centerpiece of our estimation framework is the skew-normal distribution, as proposed by Azzalini [1], which combines the normality assumption with a better fit for skewed distributions. For our estimation framework, the skew-normal is a good trade-off for a reasonably robust, yet computationally simple statistical model. The skew-normal $sn(\xi, \omega, \alpha)$ is closely related to the normal distribution $\mathcal{N}(\mu, \sigma^2)$, with an additional shape parameter α, that allows some asymmetry as skew. With the special case $sn(\mu, \sigma, 0) \sim \mathcal{N}(\mu, \sigma^2)$, the skew-normal represents a superset of the normal distribution.

To reason about computed columns, we first discuss how to fit the distribution to existing columns, before defining transformations that describe the calculation of new columns. For the base table fit, we use the *method of moments* as proposed by Pewsey [66], which uses the observed moments of a random sample. For our approach, we piggyback this calculation of the moments, in the form of descriptive statistics, onto regular table samples. To calculate these, we take a sample X of size n of each numerical column and calculate the statistics as follows:

$$\text{Mean: } \bar{x} = \frac{\sum X}{n}$$

$$\text{Standard deviation: } \bar{\sigma} = \sqrt{\frac{\sum X^2}{n} - \bar{x}^2}$$

$$\text{Skewness: } \bar{\gamma} = \frac{\frac{\sum X^3}{n} - 3\bar{x}\frac{\sum X^2}{n} + 2\bar{x}^3}{\bar{\sigma}^3}$$

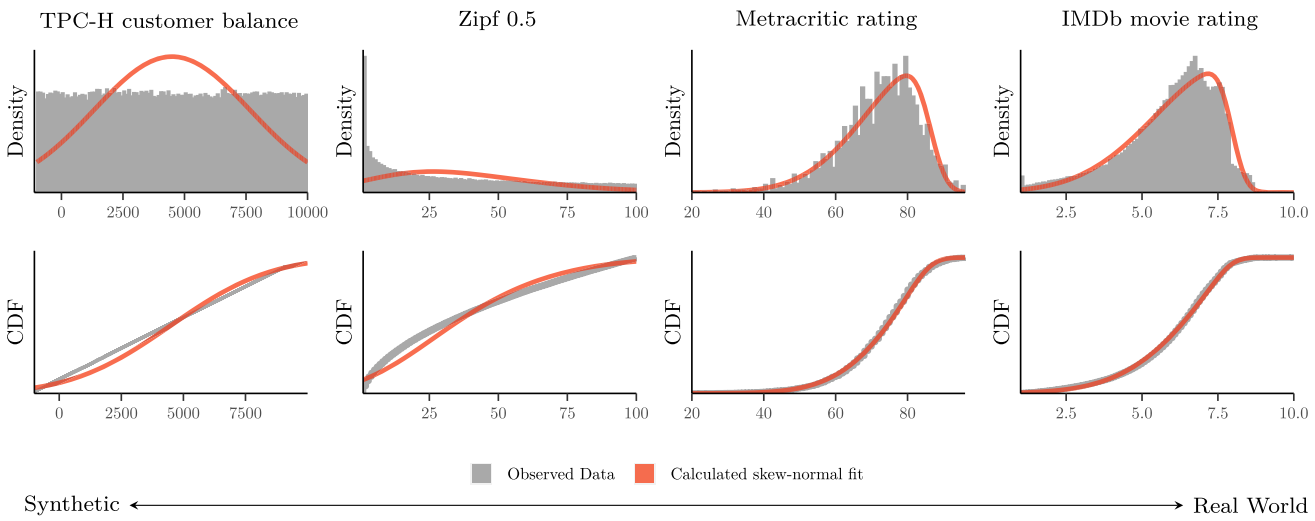


Fig. 13 Skew-normal fit. Histograms of several data sets, ranging from uniform synthetic to skewed real-world data sets. The overlayed red distribution is a fitted skew-normal distribution

Then, we transform the observed moments to the parameters of the skew-normal $sn(\xi, \omega, \alpha)$, as described by Azzalini.

$$\xi = \bar{x} - \omega \cdot m \qquad \delta = \sqrt{\pi/2} \cdot m$$
$$\omega = \sqrt{\frac{\bar{\sigma}^2}{1 - m^2}} \quad \text{where} \quad m = \frac{n}{\sqrt{1 + n^2}}$$
$$\alpha = \frac{\delta}{\sqrt{1 - \delta^2}} \qquad n = \pm\sqrt[3]{\frac{2|\bar{\gamma}|}{4 - \pi}}$$

In Umbra, we default to a sample size of 1024 values, which we keep up-to-date using reservoir sampling [7]. Our sampling process also incrementally updates the observed moments, which means that we can keep online statistics that always track the up-to-date state of the database.

Figure 13 shows the calculated skew-normal fit over four data sets. The two left distributions are both generated, i.e., uniform random data from TPC-H and a sample of a moderately skewed Zipf distribution [31]. Both distributions on the right are from real-world data sets: Steam App statistics Metacritic ratings [65] and IMDb movie ratings [49]. The figure shows the underlying data as gray histogram in the top row, and the empirical distribution function in the bottom row. Overlayed in red, we plot the PDF and the CDF, of our inferred skew-normal model.

Arguably, this method leads to a good fit of the underlying data. However, the synthetic data also pinpoints a fundamental limit of this approach. The skew-normal is unable to accurately capture the "squareness" of the uniform random data with its heavy tails, respectively "peakiness" of left edge of the Zipf distribution. More formally, the skew normal cannot fit the kurtosis—the fourth statistical moment. In addition, it can only fit a limited skewness within its parameter space ($\gamma^{\max} = \frac{\sqrt{2}(4-\pi)}{(\pi-2)^{3/2}} \approx 0.9953$ for $\alpha \to \infty$ [2]). Ideally, we would detect these edge cases and switch to a better fitting distribution using a hyperparameter model. While such a more advanced model would probably produce a better fit, the trade-off we take here has little overhead, while still fitting a CDF that produces a relatively low error for selectivity estimates of predicates.

This resulting statistical distribution $sn(\xi, \omega, \alpha)$ has several applications for our estimations. The main use-case is the estimation of $\leq$ predicates, like the one in TPC-H Q18, which follows naturally from the CDF Φ_{sn} of the skew-normal:

$$\Pr[x \leq c] = \Phi_{sn}(c)$$

Estimating equality is only possible indirectly, since the probability distribution is continuous. As approximation, we evaluate a range predicate `BETWEEN` $\pm\epsilon$ with default $\epsilon = 0.5$ to get a bucket sized for one integer.

5.2 Transformations on the skew normal

To reason about computed columns, we first define arithmetic transformations on our statistics. Given two skew normal input distributions, we model binary arithmetic expressions to estimate predicates on computed columns. As an example, consider the following condition on an analytical query that filters for orders exceeding the customer's current balance:

```
... WHERE
    part.price * ord.quantity > cust.
    balance
```

We estimate the resulting distribution of such algebraic expressions using $\circ \in \{+, -, *, /\}$ with our statistical model.

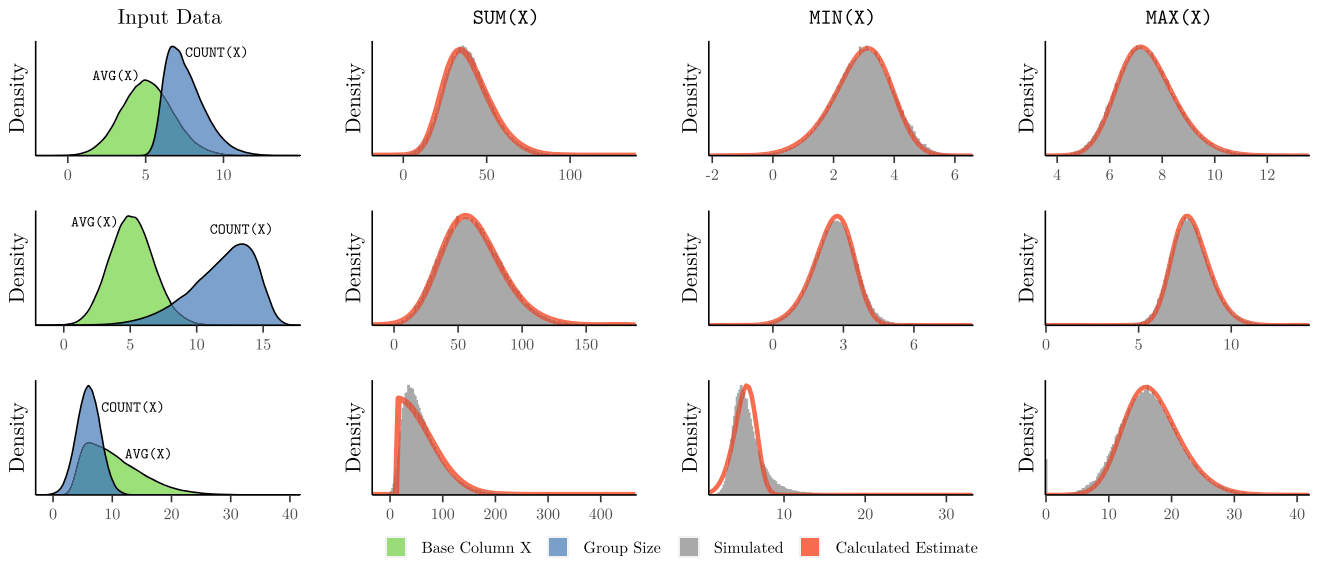


Fig. 14 Skew-normal fit of aggregates. The first column shows three different distributions of base column and group size. The next three columns compare simulated aggregates with a calculated fit of our skew-normal estimator

We piecewise transform the input moments, before fitting a skew-normal distribution for the resulting computed column:

$$\mu_{x \circ y} = \mu_x \circ \mu_y$$
$$\sigma^2_{x \circ y} = E[(x \circ y)^2] - \mu^2_{x \circ y}$$
$$\gamma_{x \circ y} = \sigma^{-3}_{x \circ y}(E[(x \circ y)^3] - 3\mu_{x \circ y}\sigma^2_{x \circ y} - \mu^3_{x \circ y})$$

5.3 Aggregate estimation

We extend these statistical building blocks on binary expressions to reason about the statistical distributions of aggregated n-ary columns. Staying with a similar example as previously, consider a query that builds an analysis on the biggest customers that have at least a revenue of one million:

```
... GROUP BY cust.id HAVING
    SUM(part.price * ord.quantity) >
    1000000
```

In the following, we go over the standard SQL aggregate functions, i.e., `AVG`, `COUNT`, `MAX`, `MIN`, and `SUM`, and discuss our estimates for these. Figure 14 shows three examples of differently skewed input columns X in green. We model the group sizes of these aggregates as i.i.d. random variables within the domain of the estimated distinct values of the grouping key [28]. This results in a binomial distribution of group sizes, which we again approximate using a skew-normal distribution, plotted in blue. For `COUNT` aggregates, this already estimates the result distribution. `AVG` aggregates are similarly independent of the group size and follow the same distribution as the input of the aggregation function.

More interesting are `SUM` aggregates, shown in the second column, which depend on both input statistical distributions: The distribution of the summed-up column, and that of the group size. We approximate the resulting computed column by a multiplication via the previously discussed transformations, and plot the resulting calculated estimate in red. To cross-validate the fit of this model, we simulate the calculation of the aggregates and plot a histogram of the resulting data in gray. For `MIN` and `MAX` aggregates, as displayed in the following two columns, we additionally need to consider their extreme value property, which we model with a Gumbel extreme value distribution G [19]. Since the distributions of maximum and minimum are symmetrical, we only detail the `MAX` case here, but `MIN` behaves similarly with flipped signs.

Let X be a skew-normal distributed random variable with inverse CDF quantile function Φ^{-1}_{sn}. Then we use the theorem of Fisher-Tippet and Gnedenko [19] to find parameters for the extreme value distribution $G(\mu, \sigma)$:

$$\mu = \Phi^{-1}_{sn}\left(1 - \frac{1}{n}\right)$$
$$\sigma = \Phi^{-1}_{sn}\left(1 - \frac{1}{n}e^{-1}\right) - \mu$$

Then, we fit a skew-normal distribution to $G(\mu, \sigma)$ to make our model closed. The resulting models provide insight on the expected distribution of such computed columns. For our query optimization pipeline, this means that we can provide accurate input for subsequent cost-based join-ordering. Good estimates, in combination with low-contention parallel execution, then produce near-optimal query plans.

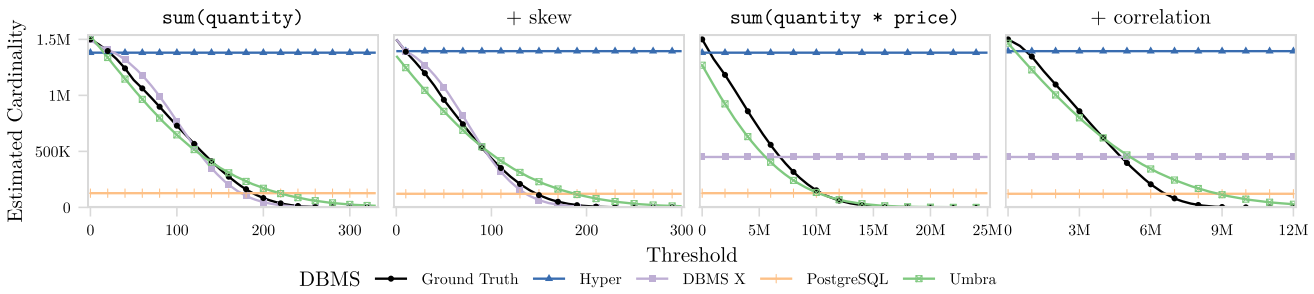


Fig. 15 Estimates for TPC-H Q18 style subqueries

6 Evaluation of estimates

In this chapter, we present the experimental evaluation of the quality of our aggregate estimations in our research RDBMS Umbra [61]. We determine how much impact improved aggregate estimates have with a comparison of the estimated cardinalities for predicates on aggregated columns. We compare our implementation to three other RDBMS, before isolating the effect of aggregate estimation.

In TPC-H, the only query with a nested aggregation is the Large Volume Customer Query Q18, with a fairly simple `HAVING` predicate. To focus on the quality of aggregate estimates, we only consider its subquery in this experiment:

```
SELECT l_orderkey
FROM lineitem
GROUP BY l_orderkey
HAVING SUM(l_quantity) > THRESHOLD
```

The subquery sums the quantity of items in an order and only selects the orders with the most numerous items. As described in the TPC-H specification, the threshold over which an order is considered large is a substitution parameter. In our experiment, we extend the range of this parameter to vary the predicate selectivity from 0% to 100% and also consider more challenging expressions.

Systems comparison In the first part of our evaluation of aggregate estimates, we consider a total of four database management systems: Tableau Hyper via its Python API 0.0.15145, DBMS X, PostgreSQL 14.4, and our research system Umbra [61]. To get accurate cardinality estimates, we load the TPC-H scale factor 1 validation data into an empty database. Then, we ensure that the DBMS has accurate statistics over this data by issuing control commands to regenerate statistics, if such commands exist. Afterwards, we execute the subquery with the substituted constant and extract the query plan. For this evaluation, we record the estimated, and the true cardinality in four different scenarios that are similar to the subquery of TPC-H Q18. First the regular Q18 aggregate over the uniform random base column `sum(quantity)`, and with a skewed Zipf 0.5 quantity. As a slightly more complex aggregate, we use `sum(quantity * price)`, again on the uniform random base columns, and with an additional anti-correlation ($\rho = -0.7$) between quantity and price (i.e., the higher the price, the lower the quantity). Note, that all these scenarios depend on group-size estimates, which we do not consider in the scope of our work, but refer to previous work [28].

Figure 15 shows this data, where the ground truth cardinality describes a decreasing curve that corresponds to higher thresholds. Our presented estimation framework in Umbra is close to the ground truth over the whole range of the threshold, even for complex predicates. In the simple scenarios with aggregates over a single column, DBMS X behaves similar. However, it does not publicly describe or document the underlying model. In addition, it falls back to estimating "magic constants" for expressions referencing more than one base column. That means, when the selectivity for a predicate cannot be determined, the systems just estimate a fixed fraction of the estimated input cardinality. Indeed, Hyper estimates 1/2 of its input estimate and PostgreSQL 1/3.

Isolating the impact of aggregate estimates We established that our estimates capture the cardinality of `HAVING` predicates. In the following, we isolate the impact of these aggregate estimations, and increase the complexity of queries and data sets. To eliminate other factors, we emulate the selectivity estimation with a fixed selectivity inside Umbra. This allows a more clear-cut evaluation of the impact of Umbra's skew-normal model on the estimation.

This evaluation uses queries on two real-world data sets. In contrast to the generated TPC-H data, these are full of correlations and non-uniform data distributions. The first data set is the Internet Movie Database (IMDb), in a slightly modified form from the Join Order Benchmark (JOB)[3]: Since IMDb primarily stores facts as strings, we extract a separate table that contains the vote count and the user rating for movies, to allow statistics collection. On these columns, we define five additional aggregation queries that calculate statistics on the new numerical columns. Furthermore, we also con-

[3] https://homepages.cwi.nl/~boncz/job/imdb.tgz.

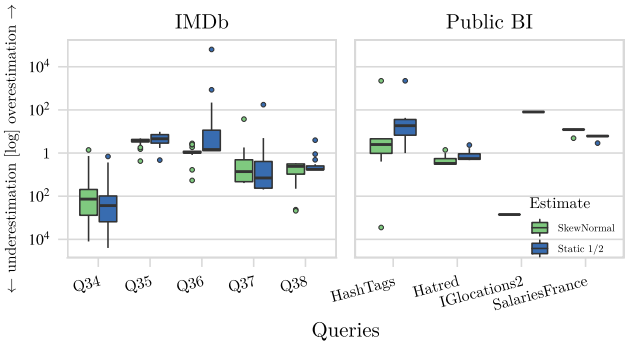


Fig. 16 Estimation quality of aggregation queries. The box plots show the log-scale q-error of our estimates in comparison to the static selectivity of Hyper. Our skew-normal model reduces the geomean q-error by 46% from 45.8 to 24.7

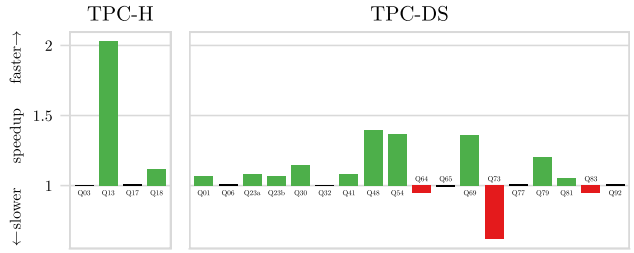


Fig. 17 Overall impact on TPC-H and TPC-DS

sider aggregation queries derived from public workbooks in Tableau Public.[4] The query set is available online.[5]

To measure the quality of the estimates, we report the *q-error*. The q-error measures the factor that an estimate differs from the ground truth. It captures the relative difference to the real value and is symmetric and multiplicative. For example, a q-error of one means that the estimate accurately captured the true cardinality, and a q-error of 10 corresponds either an over- or underestimation by a factor of 10. With a bounded q-error, it is also possible to give a theoretical guarantee about the optimal query plan quality [57].

In Fig. 16, we visualize the quality of our estimates from over 100 individual queries with predicates on aggregates. For the IMDb queries, we vary a replacement parameter of a having predicate, similar to the last experiment, to cover the whole range of 0 to 100% true selectivity. From the public BI benchmark, we consider all queries that evaluate a predicate on more than one aggregation tuple. In total, this gives us 82 IMDb aggregation queries and 48 aggregation queries from the public BI benchmark. Each box in this plot shows the median and the first and third quartiles, with individual dots for outliers.

The quality of our estimates strongly depends on the calculated statistics. For Q35 to Q38, the estimates are close to the true cardinality, with occasional outliers on the tail edges of the distribution, i.e., when the predicate is very selective. Our estimates, compared to a static selectivity estimation, capture the shape of the aggregates better and reduce over- as well as underestimation. Q34 shows one of the shortcomings of our approach, where a sum aggregate combines two distributions with heavy tails. In comparison to static selectivity estimation, our skew-normal model improves the error, but is limited by the quality of the baseline group size estimates. We found that for static estimation, we get the least error with the $1/2$ fraction that Hyper uses, which we compare in Fig. 16. In comparison to this configuration, our skew-normal selectivity estimation is a clear advantage and reduces the geometric mean of the q-errors from 45.8 to 24.7, which eliminates the impact of bad selectivity estimation.

To summarize, computed column estimates improve the estimation quality of nested aggregates. In combination with efficient parallel groupjoins, this can have significant impact on query performance. Figure 17 shows a breakdown of the affected queries in TPC-H and TPC-DS. Most queries see a moderate speedup, with only one major slowdown in TPC-DS Q73. The slowdown arises due to a worse logical plan, where previous the magic-constant estimation had a lucky guess and canceled out an unrelated error in group-size estimation. Nevertheless, it is still valuable to improve estimates so that they can capture the behavior of nested aggregates. Over the affected queries, we get a geometric mean speedup of 23% in TPC-H and 6% in TPC-DS.

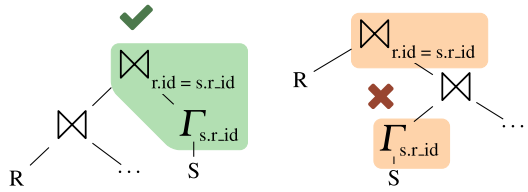
7 Planning with groupjoins

Our previous discussions of the groupjoin focussed on the implementation of an individual operator. By choosing an optimal execution strategy, we get cheaper physical execution plans. In the following, we use groupjoins to reason about the whole logical query plan and improve its overall quality.

So far, we only considered strictly better plans by opportunistically introducing groupjoins when our join reordering [64] produces a suitable plan. Specifically, this requires that the resulting plan resembles Fig. 18a, and contains a sequence of matching group-by and join without intermediary operators. Figure 18b illustrates that this misses opportunities for plans with nested aggregates. Thus, planning of groupjoins early on produces better and more robust query plans. Instead of introducing groupjoins opportunistically, we look for and eagerly introduce groupjoins for nested aggregates. This improves the plan for this potential groupjoin, and helps to find a better overall plan by providing reordering possibilities. We first show, how we eagerly introduce groupjoins in our logical query plan, before we

[4] https://github.com/cwida/public_bi_benchmark.

[5] https://db.in.tum.de/~fent/data/aggEst.tgz.



(a) Opportunistic Groupjoin **(b)** Missed Opportunity

Fig. 18 Physical planning of groupjoin operators depends on join ordering. A purely opportunistic approach misses non-trivial combinations

discuss two example queries from TPC-H that benefit from this optimization.

7.1 Eagerly introducing groupjoins

For join ordering, we build a query graph that connects relations by join predicates. For the initial construction of this graph, we consider any non-inner-joins as relations [68]. As a consequence, group-by operations form boundaries of our reordering graphs.

Algorithm 4 Identifying Groupjoins.

```
input: JoinGraph (relations, predicates)
for each GroupBy Γ ∈ relations:
   ps <- predicates joining on Γ.key
   // Precondition ①
   if ps does not cover Γ.key completely:
      continue
   R <- opposite relation(s) of ps
   // Precondition ②
   if ps is not a superkey of R:
      continue
   push (R ⋈ps Γ.key) below Γ
```

We identify potential groupjoins in this graph via Algorithm 4. The algorithm first identifies potential group-bys in the input relations and checks the preconditions to introduce a groupjoin, i.e., that we have a foreign key join with the group-by key (cf. Sect. 2.1). However, we do not immediately introduce a groupjoin, but only push this join into the group-by inputs. Swapping join and group-by allows optimizing the input even further [78]. If the join is selective, we might want to push it even further down. Keeping this conceptual groupjoin separated allows our standard reordering algorithm to determine the optimal plan. Otherwise, the join will end up at the top of the subtree, and we choose a fitting groupjoin according to our cost model (Sect. 3.5).

As a result, we get an improved intermediary plan with a conceptual groupjoin. While this plan does not necessarily result in the execution of a groupjoin, the resulting plan is strictly better than the initial plan.

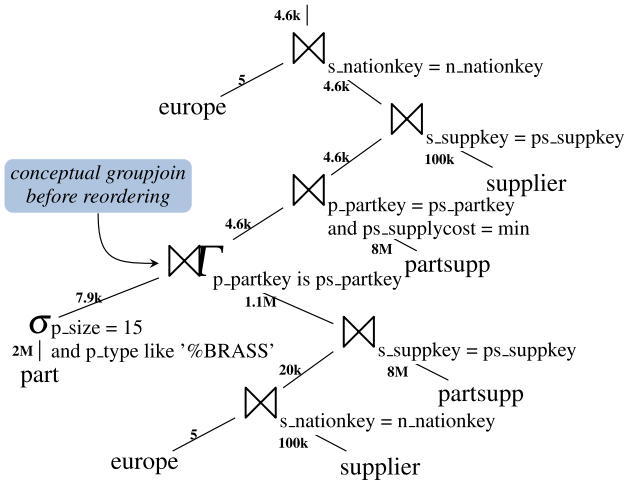


Fig. 19 Intermediary planned groupjoin for TPC-H Q2

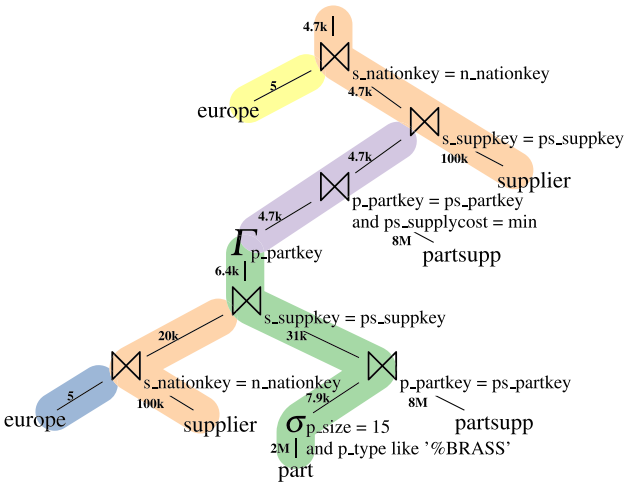


Fig. 20 Final plan for TPC-H Q2

7.2 Additional groupjoins in TPC-H

In the following, we show that this eagerly introduction of groupjoins also practically results in better plans. In TPC-H, there are two queries with such groupjoins over nested aggregates: Q2 and Q20.

TPC-H Q2 finds the minimum cost supplier for certain parts using a dependent subquery. With basic decorrelation, we calculate the minimum supply cost for all parts, before joining with the specific parts. Figure 19 shows a first execution plan to evaluate this query. Note that we abbreviated the trivial join between region and nation as the CTE "europe".

In this plan, Algorithm 4 detects a groupjoin for the nested aggregate of the join with part. Therefore, we conceptually

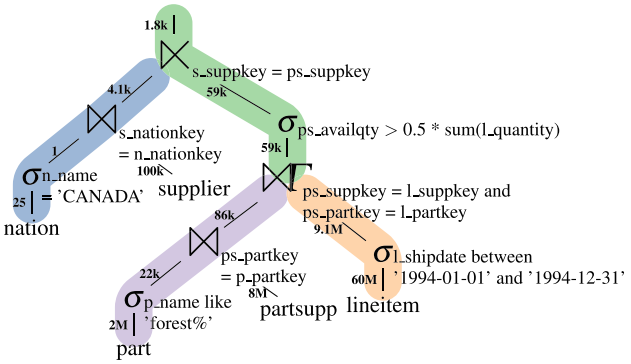


Fig. 21 Groupjoin query plan for TPC-H Q20

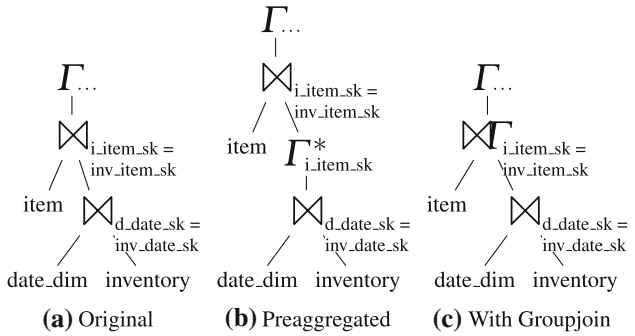


Fig. 22 Eager aggregation of TPC-DS Q22

introduce the groupjoin shown in Fig. 19. The overall utility of physically executing this groupjoin is limited, but it acts as a useful stepping stone during query planning. We then push the join below the group-by and consider it during join reordering in the lower subtree.

In a first approximation, this technique generates plans that are somewhat similar to the common optimization technique of introducing semi join reductions to filter the partsupp relation earlier [21,75]. However, since we directly join with the interesting parts, we can avoid duplicating the work of selecting the correct parts. Besides, in the final query plan shown in Fig. 20, we do not build a hash table over the parts, but use index joins, as the filtered part table is about three orders of magnitude smaller than its join partner. Thus, we can avoid the costly full table scan of the largest table in this query, partsupp, which greatly improves its runtime.

TPC-H Q20 identifies suppliers that have an excess of parts, that it determines via an aggregation over lineitem in a dependent subquery. In this subquery, we join with the partsupp relation, which we unnest initially as an outer join with the aggregate. When we collect the join graph for this query, Algorithm 4 detects that this is a groupjoin and pushes the join with part and partsupp down the aggregation. Then, we estimate the predicate on the nested aggregate using the statistical method presented in Sect. 5, which reduces the estimation q-error from 2.4 with no statistics to 1.2. Then, our join optimizer determines the optimal join order for the input of the nested aggregate, where we join early with the parts we are interested in. As a consequence, we execute the join we identified as a groupjoin last and introduce a groupjoin again, arriving at the plan shown in Fig. 21.

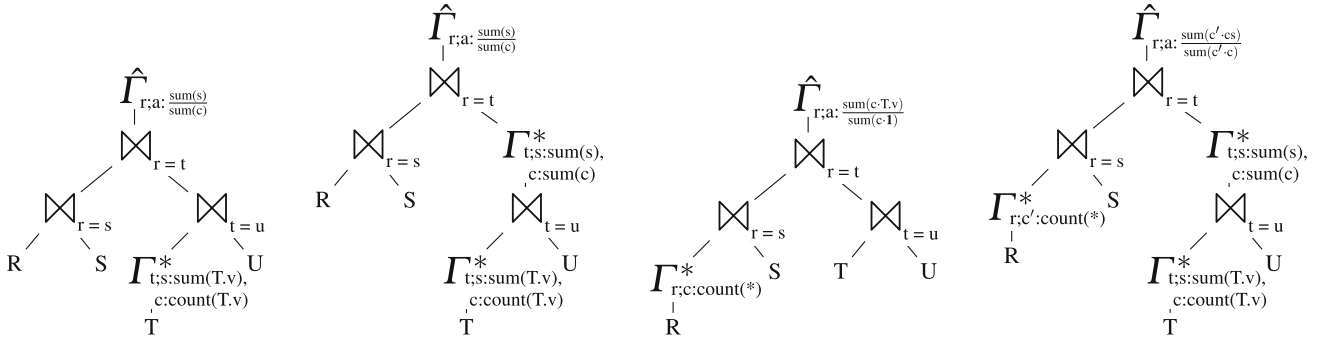
Impact In both queries, this technique aids the performance of unnested aggregates. For correlated predicates in an aggregate subquery, we transform these into a group-by key and a join over this key. When this additionally is a foreign key, our presented transformation of conceptually introducing groupjoins before reordering additionally improves these plans. Note, that this is not limited to automatically decorrelated queries, but also if the query had been flattened manually. In addition to the performance improvement already shown in the Evaluation in Sect. 4 and 6, we get additional speedups. While the two example queries from TPC-H previously had no performance changes, the changes in our query planner now result in a 67 % speedup in Q2 and a 35 % speedup in Q20.

8 Eager aggregation

In Sect. 7.1 we have explored pushing a group-by below a join in the context of groupjoins. In this section, we evaluate eager aggregations in general, pushing group-by operators further down a join tree. We propose a fast greedy approach to eager aggregation with a near-optimality guarantee, that results in significant speedups to 11 TPC-DS queries with queries 2, 22, and 59 almost reaching a 3× speedup.

Pushing a group-by down a join reduces the amount of tuples going into the join by eliminating duplicates. However, duplicates are uncommon, and eager aggregations can be costly. Thus, an eager aggregation strategy needs to intelligently identify the beneficial cases in a cost based manner.

Our eager aggregation builds on the fundamentals by Yan and Larson [78]. Chaudhuri and Shim [13] introduce a greedy heuristic for introducing eager aggregation into a join tree, however, their heuristic lacks strong guarantees on the cost of the resulting plan. We introduce an improved greedy heuristic that is guaranteed to generate plans that are optimal in cost up to a constant factor. Eich et al. [25] incorporate eager aggregations into existing dynamic programming approaches to generate optimal plans, but their approach incurs an exponential increase in optimization time. Using our greedy heuristic, near optimality can be achieved without sacrificing optimization time. In the following we summarize the fundamentals and discuss the integration of our approach into a modern optimizer. We additionally describe a cardinality estimation



(a) Using ③ **(b)** Using ② and ③ **(c)** Using ① **(d)** Using ①, ②, and ③

Fig. 23 Possible plans using the potential preaggregation placements

technique for pushed down group-by operators that allows for more consistent and reliable estimations.

As motivating example, consider the query tree of the TPC-DS Query 22 in Fig. 22a. This query first looks up the inventories of a certain date range, before finding their corresponding items, and aggregating some business metrics. However, in the end we are only interested in results *per* item. So, we are only interested in the aggregates on `i_item_sk` that we need to execute the following join, but not any particular sale or date. This means that we can preaggregate before the join with item and significantly reduce the costs of the join. Figure 22b shows the preaggreagted query tree. Note that we first group on `inv_date_sk=i_item_sk` and then group again on other attributes of item. As the newly placed group-by and the join directly above have the same hash table key we merge them into an eager groupjoin in Fig. 22c (cf. Sect. 3.1). However, this groupjoin has the slight twist that we relax Precondition ② and allow duplicates, since we aggregate them properly later.

With this strategy, we can reduce the incoming tuples into a join with an additional group-by, which we refer to as preaggregation Γ^*. To generalize this example, we first discuss when and how preaggregations can be placed to determine the space of possibilities for eager aggregation. Then we reduce this space to useful eager aggregations with a cardinality estimation strategy and a corresponding cost model. This model lends itself to a greedy approach, which we use to find near-optimal arrangements of eager aggregations.

8.1 Placement of preaggregation

There are almost no limits to where we can apply eager aggregation within a query [33]. We show this by first discussing how to represent duplicates of tuples, and how eager aggregation allows us to switch between these representations at will. Then, we study an example of the steps that we take to apply eager aggregation while guaranteeing that the resulting query is equivalent to the original query.

Relational algebra, as extended by Grefen and de By [32], works on multisets (or bags) of tuples. A multiset can contain multiple entries of the same tuple, where we refer to the amount of duplicates of a tuple as its multiplicity. Multisets can be represented practically in two main ways:

1. All duplicates are stored and processed as separate copies, i.e., the expanded representation. This represents tuple a with three duplicates and a singleton b as $\{a, a, a, b\}$
2. The multiplicity is stored within the tuple as an additional attribute m, i.e., the (strongly) aggregated representation. We represent our example tuple as $\{a^3, b^1\}$, where the superscript denotes the tuple's multiplicity.

The expanded representation of multisets is widely used, easy to implement, and performant. However, it results in repeated work for duplicates, as the same computations have to be performed for different copies of the same values. The aggregated representation allows us to eliminate such inefficiencies, but is difficult to maintain. As different operators are applied and the projection onto the required attributes shrinks, duplicates can arise, unless tuples are constantly re-aggregated after every operator. We define the weakly aggregated representation, that has both the desirable qualities of the strongly aggregated representation, but is easier to maintain. The weakly aggregated representation may contain multiple entries of the same tuple with different multiplicities. $\{a^3, b^1\}$, $\{a^2, a^1, b^1\}$, $\{a^1, a^1, a^1, b^1\}$ are all valid weakly aggregated representations of $\{a^3, b^1\}$. The multiplicity of a tuple a in a weakly aggregated representation is the sum of all the multiplicities of all the occurrences of a.

Eager aggregation allows us to intelligently switch from the expanded representation to the aggregated representation. To switch to the aggregated representation, we place a preaggregation operator [45] which can also be interpreted as a generalized projection [33]. Preaggregations should elimi-

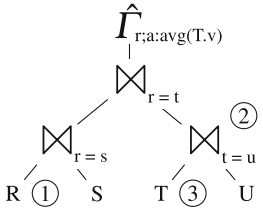


Fig. 24 Potential preaggregation placements

nate as many duplicates as possible, i.e., aggregate together as many tuples as possible, while guaranteeing that the query result is correct. When placing a preaggregation, we transform the data into the strongly aggregated representation, then do small modifications above the preaggregation to efficiently maintain a weakly aggregated representation. While we use group-bys as our preaggregation operators, a partial preaggregation operator [45] that produces tuples in weakly aggregated representation may be used analogously. If the query result, or a specific operator requires the expanded representation as input, we can use an expand-operator [33] to transform the data back to that representation.

Preaggregations can be applied in any join tree. If the result of that join tree is to be aggregated with a group-by, it is more likely that preaggregations will be useful, as the existence of the group-by in the query implies that there likely will be duplicates in the join tree's result. Thus, we will focus on applying preaggregations below a group-by, or similarly, a non-duplicate sensitive operator such as distinct, or the set operations intersect, union, and except. In these cases, an expand-operator is not even needed, as the result of the join tree can be directly processed in the (weakly) aggregated representation.

Consider the example join tree below a group-by shown in Fig. 24. Suppose our optimizer determines that duplicates are likely at points ①, ②, and ③ during the execution. Thus, we want to place preaggregation operators there to eliminate duplicates and speedup the execution. To simplify, we assume that all aggregates of $\hat{\Gamma}$ are decomposable [25], which is true for our example with the avg function.

By introducing a preaggregation operator Γ^*, we eagerly aggregate the required attributes for $\hat{\Gamma}$ in the subtree below Γ^*. All non-preaggregated attributes, e.g., $\hat{\Gamma}$'s key and join predicates, form the key of the preaggregation. When an attribute is preaggregated, it is split into three operations:

1. Initial aggregation, which computes an aggregate on attributes that have not been preaggregated.
2. Merge aggregation, which aggregates on preaggregations.
3. Finalization, computing an expression on aggregations.

For $avg(v)$, the initial aggregation is $s : sum(v)$, $c : count(v)$. In an intermediary step, we merge them: $s : sum(s)$, $c : sum(c)$. We then calculate the final aggregate as $\frac{s}{c}$. In Fig. 23a, we place a preaggregation at ③, where it calculates the initial aggregates, while $\hat{\Gamma}$ merges and finalizes them. Figure 23b shows an additional preaggregation placed in between at ②, which merges the aggregates from its input.

If we also preaggregate at ①, we need to compute and propagate the multiplicity with a $count(*)$ aggregate. The top aggregate in Fig. 23c then uses this multiplicity for the finalization of its duplicate sensitive aggregates, by multiplying their inputs with the multiplicity.

We refer to the correction of aggregates with a multiplicity as a multiplication mapping, which we also utilize for computing groupjoins in Sect. 3.1. We maintain a weakly aggregated representation of the tuples by using these multiplications. Such a multiplication is needed when both sides of a join are preaggregated. Aggregates of the left side are multiplied with the multiplicity of the right side and aggregates of the right side are multiplied with the multiplicity of the left side. The multiplicities of both sides are multiplied, resulting in the output multiplicity. Note that the multiplicity for the empty side after an outer join is 1. A multiplication is also required when only one side of a join is preaggregated, but attributes from the other side will be later aggregated above the join. Figure 23d shows an example with all preaggragations applied. In summary:

- We first determine suitable aggregates of $\hat{\Gamma}$, and include any free attributes in the preaggregation key.
- Then, we decompose the aggregates of $\hat{\Gamma}$, depending on whether the input already is preaggregated.
- Lastly, we apply multiplication mappings to maintain correct weakly aggregated representation.

For a detailed listing of decompositions for various aggregates and eager aggregation transformations, we refer to the works by Gupta et al. [33] and Eich et al. [25].

8.2 Cardinality estimation strategy for preaggregation

To find the optimal plan, join order optimizers estimate the cardinalities and costs of many plans, which means this process should be both fast and accurate. As we want our join optimizer to consider preaggregations as well, we need fast and accurate estimations for preaggregation result cardinalities. Additionally, we need our estimations to be consistent. Comparing the costs of plans with preaggregations and without *should make sense*.

To estimate the cardinality of preaggregation, one can naïvely use the standard cardinality estimator of full group-by. This approach has multiple issues. Firstly, the estimators for a full group-by are generally more involved and slower than the join size estimators used within an optimizer. Secondly, optimizers have issues meaningfully comparing join and group-by estimations as estimators tend to underestimate

joins and overestimate group-bys in many systems including but not limited to Umbra [61] and PostgreSQL. This is due to the contradictory way the independence assumption is generally applied to these two operators. The cardinality estimates of joins are usually based on the multiplication of individual selectivities of predicates, resulting in underestimations. The cardinality estimates of group-bys are usually based on the multiplication of the domain sizes of the key attributes, resulting in overestimations. These estimations are often "corrected" with a variety of heuristics, which do improve the results, but do not fundamentally change their shortcomings.

We neither want to reuse group-by estimators nor introduce a new cardinality estimator specifically for preaggregations. Regardless of how accurate such an estimator is, it will not be useful unless its estimations are sensibly comparable with join size estimations. Thus, we propose a simple strategy that relies on cardinality estimations of semi joins.

We know that $|X \bowtie_a \Gamma_a(Y)| = |X \ltimes_a Y|$ for two relations X and Y. Thus, we want to pick an estimate for $|\Gamma_a(Y)|$ in such a way that the estimate of the upcoming join cardinality $|X \bowtie_a \Gamma_a(Y)|$ would be the same as the estimate for $|X \ltimes_a Y|$. So, our estimate for $|\Gamma_a(Y)|$ is the size of Y' such that $|X \bowtie_a Y'| = |X \ltimes_a Y|$. If we use simple estimators for join and semi join based on (relative) selectivities, this results in the equivalence $|\Gamma_a(Y)| = \frac{\sigma_X}{\sigma}$, where σ is the selectivity and σ_X is the relative left selectivity of the join. This estimate fits well into a join optimizer that also considers preaggregations, as it needs to compare the cardinalities of plans of different join orderings and preaggregations. By using a common system to estimate both joins and preaggregations, we avoid introducing inconsistent estimates, which would result in the optimizer making subpar decisions.

We have shown our estimate for the case when preaggregation and join share the same key. If the preaggregation's key contains additional attributes from base relations which are not key sides of a key-foreign key join, these attributes may cause the number of distinct groups to increase. Let us consider the general case in the presence of multiple joins and additional attributes.

$\mathcal{R} :=$ the join tree of a subset of relations

$\mathcal{J} :=$ upcoming joins of the form $S \bowtie \mathcal{R}$ where $S \notin \mathcal{R}$

$\mathcal{K} :=$ the key attributes of the preaggregation

$\mathcal{A}(R) :=$ the attributes of a relation R

$dv(R_i) :=$ Estimate for $|\Gamma_{\mathcal{K} \cap \mathcal{A}(R_i)}(R_i)|$ for a relation R_i using an existing group-by estimator

Then our estimate v for $|\Gamma^*(\mathcal{R})|$ is:

$$v = \prod_{j \in \mathcal{J}} \frac{\sigma_S(j)}{\sigma(j)} \cdot \prod_{R_i \in \mathcal{R}} \begin{cases} 1, & \text{if } R_i \text{ is key side of a } j \in \mathcal{J} \\ dv(R_i), & \text{otherwise} \end{cases}$$

8.3 Integrating eager aggregation into an optimizer

Eager aggregation can theoretically speed up queries significantly, as there is no upper bound on the number of duplicates in multiset relational algebra. However, in reality, a high percentage of joins are key-foreign key joins [24] which do not produce any duplicates and prevent many eager aggregations, as any preaggregation within the key side of the join needs to contain the key, which makes such a preaggregation useless. So we need an efficient intelligent optimizer that can recognize when eager aggregations will be useful, apply eager aggregation when needed for significant gains in performance, and avoid them when they are not needed. Additionally, with the placement of eager aggregations, the optimal join orderings for a query can change as an eager aggregation can significantly reduce the cardinalities of subtrees. This further indicates the need to deeply integrate eager aggregations into a join optimizer.

Chaudhuri and Shim [13] propose a conservative (constant additional time per generated plan) and greedy (locally evaluated costs) optimization technique for eager aggregation. This technique extends an existing optimizer with an additional step that places a preaggregation directly below a join, if it locally improves the cost for that join. This is guaranteed to globally improve costs, as placing a preaggregation only decreases cardinalities and, thus, decreases costs above the preaggregation. However, this technique does not guarantee to generate optimally preaggregated plans. Instead, it only improves the (imperfect) plans of the original optimizer that does not consider preaggregations. Note that the best plan without any preaggregation can have a significantly higher cost than the optimal plan with preaggregations.

Let T^* be the globally optimal plan among all possible plans with and without preaggregations. Let J^* be the best plan without any preaggregations. In the worst case, the overhead denoted by $\frac{C(J^*)}{C(T^*)}$ can be on the order of $\mathcal{O}(n^k)$ where $C(T)$ is a cost function, relations have sizes $\mathcal{O}(n)$, and there are $O(k)$ relations. So J^* can have an exponentially higher cost than T^*. The plans generated with Chaudhuri and Shim's greedy technique is guaranteed to have a cost lower than J^*. However, this is a really high upper bound for queries where eager aggregation may be extremely useful, such as queries with multiple N-to-M joins. We propose a slightly different conservative greedy technique which is guaranteed to generate plans with costs $\mathcal{O}(C(T^*))$, meaning that our technique generates plans which are at most a constant factor larger than the globally optimal plan.

We start with the optimal query plan T^* among all possible plans, including plans with preaggregations. We take

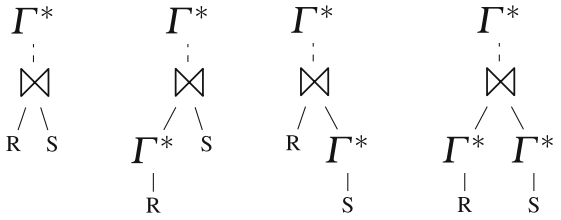


Fig. 25 Four alternative plans considered by the optimizer. All costs contain the cost of a preaggregation on the join

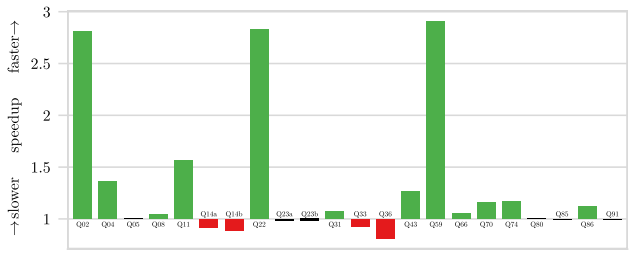


Fig. 26 Overall impact of eager aggregation on TPC-DS

this plan T^* and place additional preaggregations after every single join in a risk averse way. We call the new fully preaggregated plan $f(T^*)$. As T^* was optimal, placing these preaggregations increased the cost of the plan. However, as a preaggregation operator cannot produce more tuples than it consumes, and assuming the cost of a preaggregation operator is linear in its input size, placing additional preaggregations can only increase costs by a constant factor. Thus, $f(T^*)$ is optimal up to a constant factor. This implies that any fully preaggregated plan $f(T')$ better than the fully preaggregated optimal plan $f(T^*)$ is also optimal up to a constant factor.

It is trivial to modify a join optimizer to find the best fully preaggregated plan as the structure of fully preaggregated plans are simple; every join is always followed by a preaggregation. The optimizer should simply place a preaggregation on top of every (sub-)plan it evaluates. As we have shown, such a plan is guaranteed to be optimal up to a constant factor as it would have a lower cost than $f(T^*)$, the fully preaggregated version of the optimal plan. However, a plan with preaggregations everywhere is suboptimal. Thus, we can use a greedy approach to remove preaggregations and improve the generated plan even further. We iterate on every join, bottom-up, and remove the preaggregation on the join's left and/or right, if this locally improves the cost for the subtree including the preaggregation above this join. As we also consider the preaggregation above the join, and as the preaggregation's output size is not dependent on its input's size, this operation is guaranteed to reduce costs globally as well. The four alternative plans that are considered are shown in Fig. 25, where the join's inputs are denoted by R and S. This approach guarantees the cost upper bound $\mathcal{O}(C(T^*))$.

In our database system Umbra, we have integrated this optimization strategy into the adaptive optimization framework [64] which can compute optimal join plans for small queries using DPHyp [55] and high quality plans for larger queries using LinDP++ [68] and iterative DP [43]. With the consideration of these 4 preaggregated alternatives when building operator trees, the adaptive optimizer is able to generate high quality plans with preaggregations for a wide variety of query sizes.

Note that our approach is equivalent to the Chaudhuri and Shim [13] approach for a simple preaggregation cost function that does not depend on the input size such as $C_{bad}(\Gamma^*(T)) = \mathcal{O}(|\Gamma^*(T)|) + C(T)$. Such a cost function is not desirable as it underestimates costs and results in preaggregations being placed too eagerly. For example, in TPC-DS SF10 Query 22 as shown in Fig. 22a, an additional group-by above `inventory` is placed when C_{bad} is used. This does improve the join with `date_dim`, but not enough to be worth the group-by's cost. Thus, we use a cost function of the form $C(\Gamma^*(T)) = |T| + \mathcal{O}(|\Gamma^*(T)|) + C(T)$.

A final optimization step after the generation of preaggregation operators is to pull up these preaggregations into the joins above when possible, thus generating eager groupjoins instead of a join and a group-by. This final optimization step can result in significant performance improvements as one less pipeline needs to be generated and processed.

8.4 Evaluation of eager aggregation

We have evaluated our eager aggregation strategy on the TPC-H and TPC-DS Benchmarks. Both these benchmarks contain many join trees below group-bys. However, most of their queries are not amenable to eager aggregation, as most of the joins below group-bys are key-foreign key joins which are unlikely to produce duplicates. Thus, the eager aggregation strategy does not change the plans generated for the TPC-H queries. However, eager aggregation results in significant improvements to some queries in TPC-DS.

For TPC-DS SF 10, our optimizer places preaggregations in 22 out of 103 queries with a geometric mean speedup of around 20 % for those 22 queries. Figure 26 shows the relative speedups of individual queries. Q2, Q22, and Q59 with eager aggregation applied run more than 2.8 times as fast as their non-eagerly aggregated counterparts.

The queries with the biggest wins have a costly top group-by but simple preaggregations. Their group-bys contain multiple attributes, some of them strings, and may require additional processing for features such as `ROLLUP`. As join predicates primarily use integer keys, preaggregations below joins have integer keys as well. The costly attributes are functionally dependent on the integer keys; thus, they can be

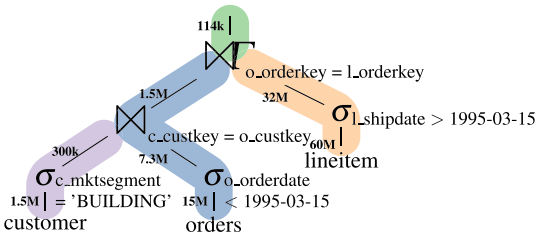


Fig. 27 Query plan for TPC-H Q3

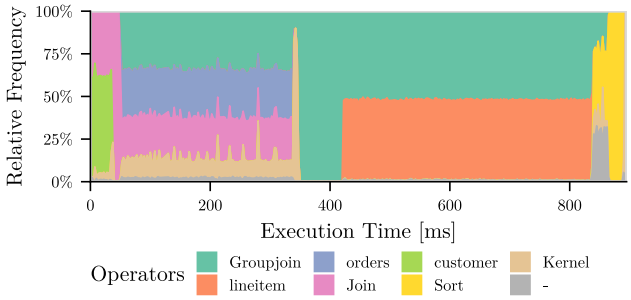


Fig. 28 Operator trace of TPC-H Q3

excluded from preaggregation keys. While our cost functions do not consider the costs of processing individual attributes, we tend to overestimate cardinalities of group-bys with larger keys, causing the optimizer to prefer simpler preaggregations.

9 Groupjoins in detail

For a qualitative analysis on the impact of groupjoins, we now take a detailed look at the queries in TPC-H that benefit from using groupjoins. In the previous sections, we already saw improved query plans. In this section, we focus on the operator selection for physical execution. Choosing the right physical execution strategy improves performance considerably, but is still sensitive to imperfect cardinality estimates.

We visualize query plans in tree form and mark our code generation pipelines with colored regions on its branches. The lowest operator of a pipeline typically generates a loop that drives the query execution. Intermediary operators then execute the query logic, before the pipelines ends at a materialization point, e.g., building a join hash table. In our interactive online query plan demonstration,[6] we use the same notation and provide additional details of intermediary optimization and cardinality estimation.

When edges between a base table and a join are not part of a pipeline, i.e., not included in a colored region, we determined it advantageous to use an index to access the table. We additionally mark all edges between operators with the cardinality of tuples that are produced on the TPC-H scale factor 10, which allows us to reason about the quality of our produced plans.

TPC-H Q3 We execute this query with a groupjoin for the top level aggregate. During optimizing of the group-by, we determine the equivalence relation between l_orderkey and o_orderkey through the join condition. Thus, we can eliminate the functionally dependent keys o_orderdate and o_shippriority, and, since we now join and aggregate over the same key, we introduce a groupjoin.

[6] https://umbra-db.com/interface/.

Figure 27 shows Umbra's execution plan for this query. In contrast to the cost model calculation in Table 3, our optimizer chooses to execute a suboptimal memoizing groupjoin. This is mainly caused by a misestimation of the relative left selectivity of the groupjoin. In this query, the filter predicates on the lineitem and order dates are correlated, but Umbra's estimations do not consider correlations [41,49]. In turn, only about a tenth of the groups in the memoizing hash table have a match, which wastes relatively much space.

The execution of this query spends most time in the groupjoin. Figure 28 shows a trace of the operator activity over the execution [5]. Note that to get a precise trace, we gathered it on a recent Ice Lake system and limited parallelism, so the total execution time is not comparable to our previous evaluation. In the plot, the relative frequency, combined with the execution time, indicates the cost of executing operators. When focussing on the groupjoin, we observe its presence in four phases. In its first phase, it works in a pipeline along the table scan of orders and the join with customer, and collects tuples in kernel allocated memory. In the second phase, it builds the hash table, before probing it with the tuples of lineitem, and scanning the results to sort and output them.

TPC-H Q13 Fig. 29 shows the physical execution plan of this query. In it, we first calculate a nested aggregate, where we count how many orders each customer has made. However, we also want to consider customers that have no recorded orders, so this query needs proper outer join semantics in the groupjoin. Since we match all customers, an eager right aggregation strategy is very advantageous.

After probing the eagerly built right aggregation hash table with the customer keys, we continue with the next aggregation that calculates the histogram over the amount of orders. This way, we directly aggregate the customers in one of the few top level groups. This pipelining, as indicated by the marked colors, keeps tuples hot and usually directly in CPU registers. In effect, we execute this query with two small, hot aggregation loops.

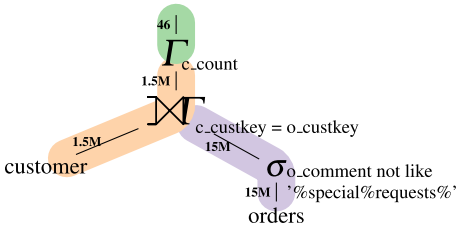


Fig. 29 Query plan for TPC-H Q13

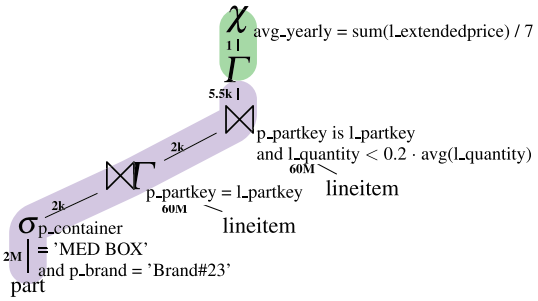


Fig. 30 Query plan for TPC-H Q17

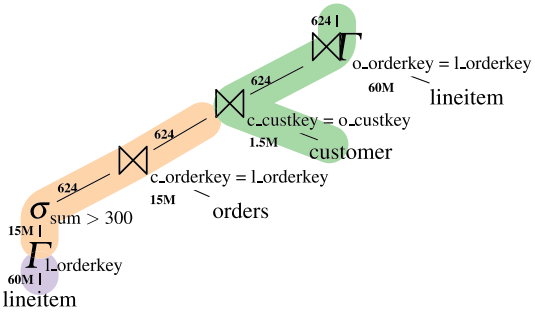


Fig. 31 Query plan for TPC-H Q18

TPC-H Q17 The query plan shown in Fig. 30, again benefits from unnesting. While we do have a static whole subquery aggregate `avg(l_quantity)`, this aggregate does not require a complex unnesting that would require all empty aggregates of the domain. Instead, we directly use all parts and join them with a groupjoin with "inner join" semantics.

Since we only have two predicates directly on the part table, we have correct estimates, and we confidently know that the qualifying parts are four orders of magnitude fewer than lineitem. Thus, we use the foreign key index on l_partkey and calculate the groupjoin aggregate pipelined, and directly continue with another probe of the same index. We then sum up the price of the matching tuples in the top level, before finally calculating the yearly average.

On a tangent, this still does some duplicate work, since we probe lineitem twice and check the partkey equivalence condition twice. As a theoretical improvement, one could execute both, groupjoin and join, with a single index lookup using the partkey. We could first calculate the aggregate for this key, before resetting and scanning the current probe index cursor again, this time checking the quantity predicate.

TPC-H Q18 This query has several interesting challenges. As we already discussed in Sect. 5, estimating the having predicate on the nested aggregate (sum > 300) is challenging. Originally, we also had a semijoin with orders as the literal translation of the `IN` expression. We transform it to an easier to execute inner join in a subsequent optimization, since we join over the key of the nested aggregate, which results in the query plan shown in Fig. 31. For the top level aggregate, we also use our knowledge of such functional dependencies of the key and drop the four functionally dependent keys, which allows the combined groupjoin with lineitem.

For physical execution, the theoretically best approach would be to use three index lookups for orders, customer, and lineitem. However, our estimates are still too uncertain, and we still estimate the left side relatively large. Umbra currently does not use in-memory optimized indexes like ART [48], but uses a more traditional disk-oriented B-tree index [4]. Since these index lookups are relatively expensive, we only use them when they are definitively faster than a full table scan. For the smaller tables, our current heuristics do not use the index, but for the large groupjoin with lineitem using the foreign key index is very advantageous.

Observations Similar to the improvements of Q2 and Q20 that we saw in Sect. 7.2, unnesting is again one of the key techniques for Q17. Since we first calculate the tiny domain, we can use a very efficient index lookup over lineitem to calculate the groupjoin result.

Using index lookups to answer queries is often very profitable, but one limiting factor for them are estimation errors. For example, in Q18, we use a suboptimal query plan that does not use the indexes of orders and customer due to our low confidence in the estimation of the nested aggregate. While our aggregate estimates from Sect. 5 already improve these, we still estimate that several thousand sums qualify the condition, and we only choose to use the index for the largest join partner, lineitem.

Using the index for the groupjoin in Q17, results in a 6 × speedup for scale factor 10. In Q18, the shown query plan has an 15 % speedup compared to using a memoizing groupjoin. If we estimate the having predicate accurately and correctly identify that few tuples qualify, we would also use an index for customer, however the performance impact is negligible.

The tracking and inference of functional dependencies is another fundamental optimization for these queries. These allow us to minimize the grouping keys to only the candidate key over which we join with another table. In TPC-DS, we can use significantly fewer groupjoins, since the functional dependencies are not directly included in the schema.

Although the DS schema defines primary keys, it additionally defines so-called business keys, which we cannot represent in the schema. Many aggregates then use the business keys, where we miss the functional dependency to minimize the keys and thus do not produce optimal execution plans.

10 Related work

As outlined in Sect. 2, our work relies on well-known work on query unnesting, which enables aggregates to be embedded in the query tree [18,27,39,62]. Subquery unnesting to flatten the query tree is well-known as one of the most important aspects of query optimization [10,21]. Galindo-Legaria and Joshi [29] describe the comprehensive optimization of aggregation in Microsoft SQL Server. They describe the reordering of group-by and outer-join, where they use similar conditions to our groupjoin preconditions (cf. Sect. 2.1) and also discuss the problems with `COUNT` in static (scalar) aggregation. In contrast to our work on groupjoins, they keep join and aggregation separate, where a pushed-down group-by will still build a redundant hash table.

Bellamkonda et al. [6] describe the execution of correlated subqueries with window operations in Oracle. Hölsch et al. [35] use an extended form of relational algebra to reason about nested queries and are able to express more transformation on aggregations. To incorporate unnested aggregations in cost models, practical implementations, e.g., in DB2, use statistical views [26]. However, each query needs a matching view, which are relatively costly to create and maintain, and are usually only created where missing statistics lead to very poor plans.

In many real-world evaluations, join and aggregation are big contributors to the overall workload [37,77]. Consequently, there is a large body of related work that optimizes hash joins [52,63,71] and hash aggregations [51,67,80]. Re-using hash partitions, and even whole hash tables is a well-known optimization [18,36]. One often discussed question is, if hash tables should be partitioned or non-partitioned [3]. Our proposed approaches in Sect. 3 try to use a non-partitioned hash table to avoid materializing data, while using thread-local partitioning for heavy-hitters. Other recent work on the interaction of multiple operators focused on memory access patterns to better utilize the available hardware [16,54]. We see this work as orthogonal, and these ideas can work hand-in-hand with parallel groupjoin execution.

11 Conclusion

In this paper, we improved several aspects for an efficient evaluation of joins and aggregates in a general-purpose relational database management system. We improve important pieces of the query engine that previously have not worked well with nested aggregates. First, we presented a low overhead estimation of computed columns, which significantly improves the estimates that we use to find better query plans in the query optimizer. Our aggregate estimates result in a near 50% reduction of estimation error, without any changes to the underlying sampling method.

Furthermore, we improved the execution of groupjoins, which commonly occur in nested and regular aggregation queries. Our contention-free parallel and index-based execution allows them to be more universally applicable. We also demonstrated where using a groupjoin is advantageous and presented a simple, yet effective cost model to plan the best execution strategy. With our improved groupjoin execution, we achieve significant speedups in several TPC-H queries.

Building on our improvements to groupjoins, we presented an eager aggregation technique that significantly improves execution plans with minimal regressions. It consists of a simple cardinality estimation strategy, and a novel greedy conservative optimization approach to introducing preaggregation operators. When integrated into the adaptive optimization framework, this technique introduces preaggregations in 22 TPC-DS queries, resulting in approximately 20 % geometric mean speedup with scale factor 10, with 3 queries reaching almost a 3 × speedup.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Azzalini, A.: A class of distributions which includes the normal ones. Scand. J. Stat. **12**, 171–178 (1985)
2. Azzalini, A.: The Skew-Normal and Related Families, vol. 3. Cambridge University Press (2013)
3. Bandle, M., Giceva, J., Neumann, T.: To partition, or not to partition, that is the join question in a real system. In: SIGMOD, pp. 168–180. ACM (2021)
4. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. In: SIGFIDET Workshop, pp. 107–141. ACM (1970)

5. Beischl, A., Kersten, T., Bandle, M., Giceva, J., Neumann, T.: Profiling dataflow systems on multiple abstraction levels. In: EuroSys, pp. 474–489. ACM (2021)
6. Bellamkonda, S., Ahmed, R., Witkowski, A., Amor, A., Zaït, M., Lin, C.C.: Enhanced subquery optimizations in Oracle. Proc. VLDB Endow. **2**(2), 1366–1377 (2009)
7. Birler, A., Radke, B., Neumann, T.: Concurrent online sampling for all, for free. In: DaMoN, pp. 5:1–5:8. ACM (2020)
8. Boehm, H., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI, pp. 68–78. ACM (2008)
9. Boncz, P.A., Anadiotis, A.G., Kläbe, S.: JCC-H: adding join crossing correlations with skew to TPC-H. In: TPCTC, Lecture Notes in Computer Science, vol. 10661, pp. 103–119. Springer (2017)
10. Boncz, P.A., Neumann, T., Erling, O.: TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In: TPCTC, Lecture Notes in Computer Science, vol. 8391, pp. 61–76. Springer (2013)
11. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB X100: Hyper-pipelining query execution. In: CIDR (2005)
12. Chatziantoniou, D., Akinde, M.O., Johnson, T., Kim, S.: The MD-join: an operator for complex OLAP. In: ICDE, pp. 524–533. IEEE (2001)
13. Chaudhuri, S., Shim, K.: Including group-by in query optimization. In: VLDB, pp. 354–366 (1994)
14. Cieslewicz, J., Ross, K.A.: Adaptive aggregation on chip multiprocessors. In: VLDB, pp. 339–350. ACM (2007)
15. Cluet, S., Moerkotte, G.: Efficient evaluation of aggregates on bulk types. In: DBPL, Electronic Workshops in Computing, p. 8. Springer (1995)
16. Crotty, A., Galakatos, A., Kraska, T.: Getting swole: generating access-aware code with predicate pullups. In: ICDE, pp. 1273–1284. IEEE (2020)
17. Dageville, B., Cruanes, T., Zukowski, M., Antonov, V., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., Lee, A.W., Motivala, A., Munir, A.Q., Pelley, S., Povinec, P., Rahn, G., Triantafyllis, S., Unterbrunner, P.: The Snowflake elastic data warehouse. In: SIGMOD, pp. 215–226. ACM (2016)
18. Dayal, U.: Of nests and trees: a unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In: VLDB, pp. 197–208 (1987)
19. De Haan, L., Ferreira, A.: Extreme Value Theory: An Introduction. Springer (2007)
20. Diaconu, C., Freedman, C., Ismert, E., Larson, P., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL server's memory-optimized OLTP engine. In: SIGMOD, pp. 1243–1254. ACM (2013)
21. Dreseler, M., Boissier, M., Rabl, T., Uflacker, M.: Quantifying TPC-H choke points and their optimizations. Proc. VLDB Endow. **13**(8), 1206–1220 (2020)
22. Durner, D., Leis, V., Neumann, T.: On the impact of memory allocation on high-performance query processing. In: DaMoN, pp. 21:1–21:3. ACM (2019)
23. Dutt, A., Wang, C., Nazi, A., Kandula, S., Narasayya, V.R., Chaudhuri, S.: Selectivity estimation for range predicates using lightweight models. Proc. VLDB Endow. **12**(9), 1044–1057 (2019)
24. Eich, M.: Extending dynamic-programming-based plan generators: beyond pure enumeration. Ph.D. thesis, University of Mannheim, Germany (2017)
25. Eich, M., Fender, P., Moerkotte, G.: Efficient generation of query plans containing group-by, join, and groupjoin. VLDB J. **27**(5), 617–641 (2018)
26. El-Helw, A., Ilyas, I.F., Zuzarte, C.: StatAdvisor: recommending statistical views. Proc. VLDB Endow. **2**(2), 1306–1317 (2009)
27. Elhemali, M., Galindo-Legaria, C.A., Grabs, T., Joshi, M.: Execution strategies for SQL subqueries. In: SIGMOD, pp. 993–1004. ACM (2007)
28. Freitag, M.J., Neumann, T.: Every row counts: combining sketches and sampling for accurate group-by result estimates. In: CIDR (2019)
29. Galindo-Legaria, C.A., Joshi, M.: Orthogonal optimization of subqueries and aggregation. In: SIGMOD, pp. 571–581. ACM (2001)
30. Graefe, G., Kuno, H.A.: Modern B-tree techniques. In: ICDE, pp. 1370–1373. IEEE (2011)
31. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: SIGMOD (1994)
32. Grefen, P.W.P.J., de By, R.A.: A multi-set extended relational algebra—a formal approach to a practical issue. In: ICDE, pp. 80–88. IEEE (1994)
33. Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. In: VLDB, pp. 358–369 (1995)
34. Hirn, D., Grust, T.: PgCuckoo: Laying plan eggs in PostgreSQL's nest. In: SIGMOD, pp. 1929–1932. ACM (2019)
35. Hölsch, J., Grossniklaus, M., Scholl, M.H.: Optimization of nested queries using the NF2 algebra. In: SIGMOD, pp. 1765–1780. ACM (2016)
36. Kemper, A., Kossmann, D., Wiesner, C.: Generalised hash teams for join and group-by. In: VLDB, pp. 30–41 (1999)
37. Kersten, M., Koutsourakis, P., Nes, N., Zhan, Y.: Bridging the chasm between science and reality. In: CIDR (2021)
38. Kim, C., Sedlar, E., Chhugani, J., Kaldewey, T., Nguyen, A.D., Blas, A.D., Lee, V.W., Satish, N., Dubey, P.: Sort versus hash revisited: fast join implementation on modern multi-core CPUs. Proc. VLDB Endow. **2**(2), 1378–1389 (2009)
39. Kim, W.: On optimizing an SQL-like nested query. ACM Trans. Database Syst. **7**(3), 443–469 (1982)
40. Kipf, A., Freitag, M., Vorona, D., Boncz, P., Neumann, T., Kemper, A.: Estimating filtered group-by queries is hard: Deep learning to the rescue. In: AIDB (2019)
41. Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P.A., Kemper, A.: Learned cardinalities: Estimating correlated joins with deep learning. In: CIDR (2019)
42. Kohn, A., Leis, V., Neumann, T.: Building advanced SQL analytics from low-level plan operators. Proc. VLDB Endow. **14** (2021)
43. Kossmann, D., Stocker, K.: Iterative dynamic programming: a new class of query optimization algorithms. ACM Trans. Database Syst. **25**(1), 43–82 (2000)
44. Lang, H., Mühlbauer, T., Funke, F., Boncz, P.A., Neumann, T., Kemper, A.: Data blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: SIGMOD, pp. 311–326. ACM (2016)
45. Larson, P.: Data reduction by partial preaggregation. In: ICDE, pp. 706–715. IEEE (2002)
46. Leis, V., Boncz, P.A., Kemper, A., Neumann, T.: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: SIGMOD, pp. 743–754. ACM (2014)
47. Leis, V., Gubichev, A., Mirchev, A., Boncz, P.A., Kemper, A., Neumann, T.: How good are query optimizers, really? Proc. VLDB Endow. **9**(3), 204–215 (2015)
48. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: ICDE, pp. 38–49. IEEE (2013)
49. Leis, V., Radke, B., Gubichev, A., Mirchev, A., Boncz, P.A., Kemper, A., Neumann, T.: Query optimization through the looking glass, and what we found running the join order benchmark. VLDB J. **27**(5), 643–668 (2018)
50. Leis, V., Scheibner, F., Kemper, A., Neumann, T.: The ART of practical synchronization. In: DaMoN, pp. 3:1–3:8. ACM (2016)
51. Liu, F., Salmasi, A., Blanas, S., Sidiropoulos, A.: Chasing similarity: distribution-aware aggregation scheduling. Proc. VLDB Endow. **12**(3), 292–306 (2018)

52. Makreshanski, D., Giannikis, G., Alonso, G., Kossmann, D.: Many-query join: efficient shared execution of relational joins on modern hardware. VLDB J. **27**(5), 669–692 (2018)
53. May, N., Moerkotte, G.: Main memory implementations for binary grouping. In: XSym, Lecture Notes in Computer Science, vol. 3671, pp. 162–176. Springer (2005)
54. Menon, P., Pavlo, A., Mowry, T.C.: Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. Proc. VLDB Endow. **11**(1), 1–13 (2017)
55. Moerkotte, G., Neumann, T.: Faster join enumeration for complex queries. In: ICDE, pp. 1430–1432. IEEE (2008)
56. Moerkotte, G., Neumann, T.: Accelerating queries with group-by and join by groupjoin. PVLDB **4**(11), 843–851 (2011)
57. Moerkotte, G., Neumann, T., Steidl, G.: Preventing bad plans by bounding the impact of cardinality estimation errors. Proc. VLDB Endow. **2**(1), 982–993 (2009)
58. Muralikrishna, M.: Improved unnesting algorithms for join aggregate SQL queries. In: VLDB, pp. 91–102 (1992)
59. Nambiar, R.O., Poess, M.: The making of TPC-DS. In: VLDB, pp. 1049–1058. ACM (2006)
60. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. **4**(9), 539–550 (2011)
61. Neumann, T., Freitag, M.J.: Umbra: a disk-based system with in-memory performance. In: CIDR (2020)
62. Neumann, T., Kemper, A.: Unnesting arbitrary queries. In: BTW, LNI, vol. P-241, pp. 383–402. GI (2015)
63. Neumann, T., Leis, V., Kemper, A.: The complete story of joins (in HyPer). In: BTW, LNI, vol. P-265, pp. 31–50. GI (2017)
64. Neumann, T., Radke, B.: Adaptive optimization of very large join queries. In: SIGMOD, pp. 677–692. ACM (2018)
65. O'Neill, M., Vaziripour, E., Wu, J., Zappala, D.: Condensing Steam: Distilling the diversity of gamer behavior. In: Internet Measurement Conference, pp. 81–95. ACM (2016)
66. Pewsey, A.: Problems of inference for Azzalini's skewnormal distribution. J. Appl. Stat. **27**(7), 859–870 (2000)
67. Polychroniou, O., Ross, K.A.: High throughput heavy hitter aggregation for modern SIMD processors. In: DaMoN, p. 6. ACM (2013)
68. Radke, B., Neumann, T.: LinDP++: generalizing linearized DP to crossproducts and non-inner joins. In: BTW, LNI, vol. P-289, pp. 57–76. GI (2019)
69. Raman, V., Attaluri, G.K., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G.M., Malkemus, T., Müller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A.J., Zhang, L.: DB2 with BLU acceleration: so much more than just a column store. Proc. VLDB Endow. **6**(11), 1080–1091 (2013)
70. Schleich, M., Olteanu, D., Khamis, M.A., Ngo, H.Q., Nguyen, X.: A layered aggregate engine for analytics workloads. In: SIGMOD, pp. 1642–1659. ACM (2019)
71. Schuh, S., Chen, X., Dittrich, J.: An experimental comparison of thirteen relational equi-joins in main memory. In: SIGMOD, pp. 1961–1976. ACM (2016)
72. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD, pp. 23–34. ACM (1979)
73. Seshadri, P., Hellerstein, J.M., Pirahesh, H., Leung, T.Y.C., Ramakrishnan, R., Srivastava, D., Stuckey, P.J., Sudarshan, S.: Cost-based optimization for magic: algebra and implementation. In: SIGMOD, pp. 435–446. ACM Press (1996)
74. Steenhagen, H.J.: Optimization of object query languages. Ph.D. thesis, University of Twente (1995)
75. Stocker, K., Kossmann, D., Braumandl, R., Kemper, A.: Integrating semi-join-reducers into state of the art query processors. In: ICDE, pp. 575–584. IEEE (2001)
76. Stonebraker, M., Rowe, L.A.: The design of Postgres. In: SIGMOD, pp. 340–355. ACM Press (1986)
77. Vogelsgesang, A., Haubenschild, M., Finis, J., Kemper, A., Leis, V., Mühlbauer, T., Neumann, T., Then, M.: Get real: how benchmarks fail to represent the real world. In: DBTest@SIGMOD, pp. 1:1–1:6. ACM (2018)
78. Yan, W.P., Larson, P.: Performing group-by before join. In: ICDE, pp. 89–100. IEEE (1994)
79. Yan, W.P., Larson, P.: Eager aggregation and lazy aggregation. In: VLDB, pp. 345–357 (1995)
80. Zhang, Z., Deshmukh, H., Patel, J.M.: Data partitioning for in-memory systems: myths, challenges, and opportunities. In: CIDR (2019)