

A Practical Approach to Groupjoin and Nested Aggregates

Philipp Fent

Technische Universität München
fent@in.tum.de

Thomas Neumann

Technische Universität München
neumann@in.tum.de

ABSTRACT

Groupjoins, the combined execution of a join and a subsequent group by, 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, which limits their usefulness in a many-core system. Having an efficient implementation of groupjoins is highly desirable, as groupjoins are not only used to fuse group by and join but are also introduced by the unnesting component of the query optimizer to avoid nested-loops evaluation of aggregates. Furthermore, the query optimizer needs to be able to reason over the result of aggregation in order to schedule it correctly. Traditional selectivity and cardinality estimations quickly reach their limits when faced with computed columns from nested aggregates, which leads to poor cost estimations and thus, suboptimal query plans.

In this paper, we present techniques to efficiently estimate, plan, and execute groupjoins and nested aggregates. We propose two novel techniques, *aggregate estimates* to predict the result distribution of aggregates, and *parallel groupjoin execution* for a scalable execution of groupjoins. The resulting system has significantly better estimates and a contention-free evaluation of groupjoins, which can speed up some TPC-H queries up to a factor of 2.

PVLDB Reference Format:

Philipp Fent and Thomas Neumann. A Practical Approach to Groupjoin and Nested Aggregates. PVLDB, 14(11): 2383 - 2396, 2021.

doi:10.14778/3476249.3476288

1 INTRODUCTION

Joins and aggregations are the backbone of query engines. A common query pattern, which we observe in many benchmarks [7, 45] and industry applications [58], 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* [42].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.

doi:10.14778/3476249.3476288

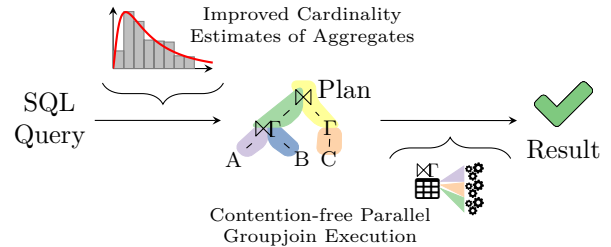


Figure 1: Missing components for practical groupjoins. Our improvements to estimation and parallel execution enable efficient evaluation of queries with nested aggregates.

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
)
```

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` [44]. A groupjoin directly supports such 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 the groupjoin hash table contains 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 [37]. With unnested aggregates, the estimation includes group-by operations and aggregates, which are notoriously hard [22, 32]. 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 [54, 59] and hash table sharing via groupjoins [19, 42]. 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 [47]. 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 Section 2. Then, we discuss and evaluate three parallel groupjoin execution strategies in Section 3, and propose a cost model to choose the optimal execution strategy. Afterwards, we introduce our estimations for computed columns in Section 4. Section 5 shows our experimental results based on the well-known TPC-H and TPC-DS benchmarks, before we discuss related work in Section 6.

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 [4, 26, 48]. However, unnested and decorrelated aggregates need a careful implementation and are challenging in further query planning.

2.1 Groupjoin

We define a groupjoin \bowtie^{Γ} [42] as an equi-join with separate aggregates over its binary inputs grouped by the join key.

$$R \bowtie^{\Gamma}_{a_1 = a_2} \text{agg } S := \{r \circ [g_r : G_R] \circ [g_s : G_S] \mid r \in R, \\ G_S = \text{agg}(\{s \mid s \in S \wedge r.a_1 = s.a_2\}), \\ G_R = \text{agg}(\{r \mid r \in R \wedge r.a_1 = s.a_2\})\}$$

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

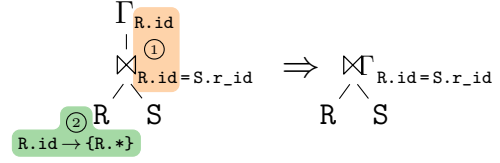


Figure 2: Preconditions to introduce a groupjoin.

Figure 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 a superkey w.r.t. functional dependencies of the left build side. In this case, introducing a groupjoin is usually considered to be a net win [14, 19] 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 Section 1, where we calculate a whole-table $\text{COUNT}(\ast)$ 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 $\text{COUNT}(\ast)$ example, we initialize empty groups (e.g., customers with no sales) as zero, and increment it with whole-table tuple counting logic¹.

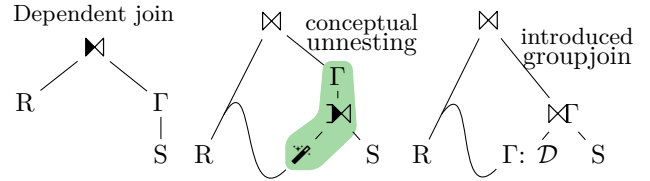


Figure 3: General Unnesting: Decorrelation of dependent subqueries containing an aggregation can introduce a groupjoin.

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 [48]. Figure 3 shows this unnesting for two arbitrary tables R and S , with the dependent subquery-join \bowtie on top and a nested whole table aggregate Γ in the correlated right subtree. To decorrelate this aggregate, we first compute the magic set \mathcal{Z} of relevant tuples for the correlated subquery [56]. 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

¹ $\text{COUNT}(\ast)$ has some edge cases that are trivial in a groupjoin, but difficult in separate operators. See Section 3.3 for an extended discussion.

both preconditions to introduce a groupjoin, which we then 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 [11, 30, 49]. 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 [9, 12, 40].

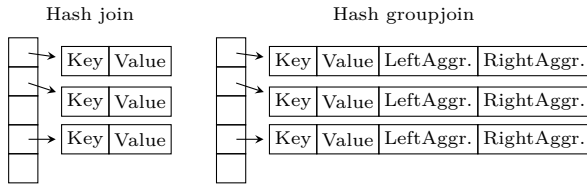


Figure 4: Single Threaded Groupjoin Hash Table. Aggregates from either join side are materialized as hash table payload.

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 [52]. 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. 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 Section 3.4, 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 [59]. 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.

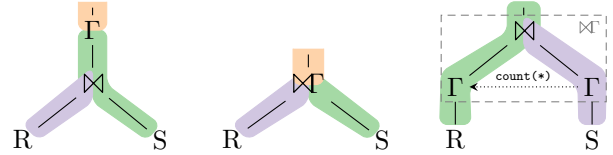


Figure 5: Eager Grouping. While the middle groupjoin eliminates the second hash table, the schematic eager aggregation on the right can additionally eliminate the result scan.

When eagerly aggregating in a groupjoin, we can exploit several facts that allow making eager aggregation very efficient: Precondition ② (cf. Section 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 Figure 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 [59], and replace the result scan with a single hash table probe.

Algorithm 1: Example code generated to execute an eager right groupjoin.

```

initialize memory of  $\Gamma_S$ 
for each tuple  $s$  in  $S$ 
    aggregate  $s$  as  $a_s$  in hash table of  $\Gamma_S$ 
for each tuple  $r$  in  $R$ 
    if  $r$  has match  $a_s$  in  $\Gamma_S$ 
         $a_r := \text{agg}(r * a_s.\text{count}(*))$ 
    output:  $r \circ a_r \circ a_s$ 

```

We eliminate the result scan, and improve the pipeline behavior, by using the same precondition ②. A group-by is a *full pipeline breaker* [46], 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.

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 [34, 35, 53]. 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:

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

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 $|R \times S|$, 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 don't need to materialize their result.

3.2 Memoizing Groupjoin

Eagerly aggregating is very beneficial, when every right tuple finds a join partner. The other extreme is also common, 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 Figure 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.

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 [6]. For our optimistic synchronization, we use a

Algorithm 2: Memoizing groupjoin probe pipeline with ownership tagging.

```

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

```

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 two-pass build of the left hash table $2|R|$, probe once with the entire right side $|S|$, before checking the matching tuples $|R \times S|$ for equality. Then, we use these to build thread-local aggregates, before merging them into their memoized global bucket, $2|R \times S|$. 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:

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

3.3 Separating Join and Group-By

As laid out in Section 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 [14, 19, 42] 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-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 that we need to unnest arbitrary queries (cf. Section 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 [12]. 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 $\text{count}(\ast)$ in a subquery counts any matching tuple, even when its value is NULL. Executing an outer join $R \bowtie S$, 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
```

Table 1: Static count semantics. In separate operators $\text{count}(\ast)$ aggregates might produce different results.

$R \bowtie S$	$\text{count}(\ast)$ subquery	$\text{count}(S)$ after \bowtie	$\text{count}(\ast)$ before \bowtie
(NULL, NULL)	1	0	1
(1, 1)	1	1	1
(2, NULL)	0	0	NULL

```
) 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 $|R \times S|$ right tuples. With the resulting tuples, we build a separate aggregation table, again in two passes $2|R \times S|$, before we scan the $|R \times S|$ 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 ($|R \times S|$) 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:

$$C_{\text{separate}} = |R| + (3 + c) |R \times S| + |R \times S|$$

3.4 Choosing a Physical Implementation

To recap, we presented three parallel execution strategies for groupjoins. In Section 3.1, we presented an eagerly right aggregating groupjoin, in Section 3.2 we use a combined join and aggregation table with memoizing thread-local aggregations. Lastly, we showed in Section 3.3, that we can rewrite arbitrary groupjoins as separate join and group-by. All three implementations have different characteristics, which we formalized as a cost model to compare their relative performance:

$$\begin{aligned} C_{\text{eager}} &= |S| + |R \times S| \\ C_{\text{memo}} &= 2|R| + 3|R \times S| \\ C_{\text{separate}} &= |R| + 3.3|R \times S| + |R \times S| \end{aligned}$$

Table 2: Example cost calculations of groupjoin implementations.

$ R $	$ S $	σ_R	σ_S	$ R \times S $	$ R \bowtie 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

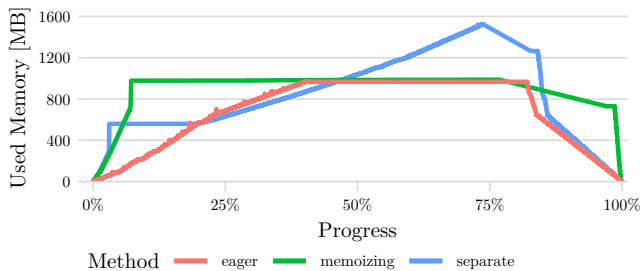


Figure 6: Memory consumption of TPC-H SF 10 orders - lineitem groupjoins.

The base of all three cost functions consists of the underlying cardinalities $|R|$ and $|S|$, and the semijoin reduced cardinalities $|R \bowtie S| = |R| \sigma_R$ and $|R \bowtie S| = |S| \sigma_S$. 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 $C_{separate}$ 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 1 GB 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 $C_{separate}$. 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.

4 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 [36, 49]. 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

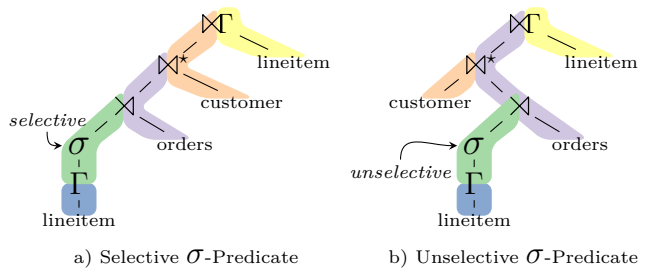


Figure 7: Possible query plans for TPC-H Query 18. Depending on the σ filter selectivity, we use the customer relation as hash-join build or probe side, which roughly leads to a 10 % difference in performance.

multi-column estimates [18, 33]. However, estimates for computed columns such as aggregates are rarely used, which results in poor cost-model calculations, and in turn 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 Figure 7, 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^* , 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 (150 k customers and 1.6 M 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 0.003 %, we need robust estimation of computed columns.

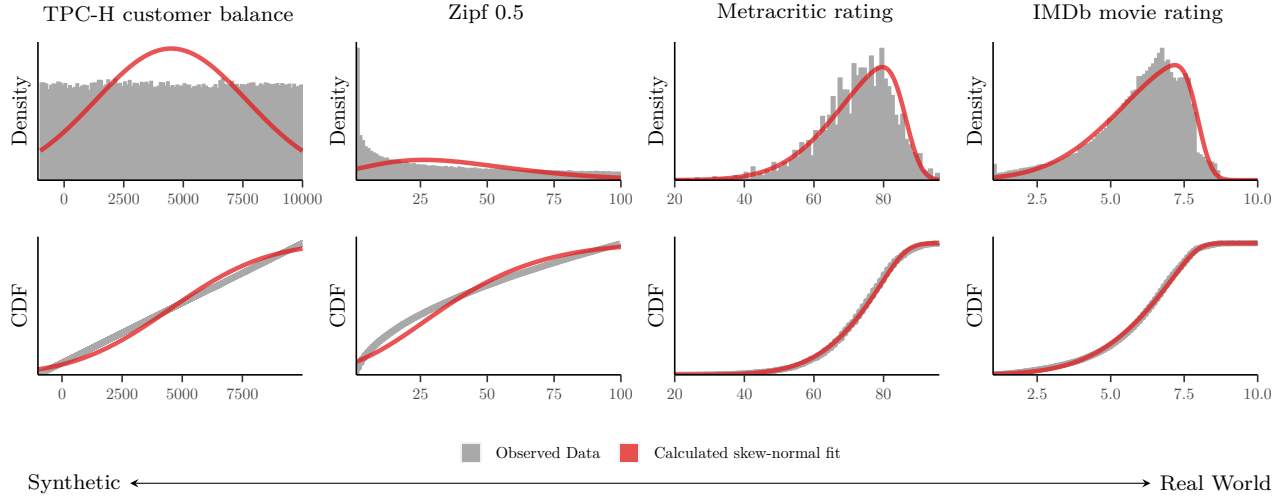


Figure 8: Skew-Normal Fit. Histograms of several data sets, ranging from uniform synthetic to skewed real-world data sets. The overlaid red distribution is a fitted skew-normal distribution.

In the following, we present our novel computed column estimator, based on the method of moments for skew-normal distributions [51]. In result, we get orders of magnitude better estimates for filters on computed columns and in turn generate better query plans.

4.1 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 [51], 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:

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

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

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

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

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

In Umbra, we default to a sample size of 1024 values, which we keep up-to-date using reservoir sampling [5]. 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 8 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 [24]. Both distributions on the right are from real-world data sets: Steam App statistics Metacritic ratings [50] and IMDb movie ratings [37]. The figure shows the underlying data as gray histogram, and the empirical distribution

function in the bottom row. Overlaid 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 $\alpha \rightarrow \infty$ [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.

4.2 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. First, we piecewise transform the input moments, before fitting a skew-normal distribution for the resulting computed column as follows:

$$\begin{aligned} \mu_{x \circ y} &= \mu_x \circ \mu_y \\ \sigma_{x \circ y}^2 &= E[(x \circ y)^2] - \mu_{x \circ y}^2 \\ \gamma_{x \circ y} &= \sigma_{x \circ y}^{-3} (E[(x \circ y)^3] - 3\mu_{x \circ y} \sigma_{x \circ y}^2 - \mu_{x \circ y}^3) \end{aligned}$$

4.3 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 9 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 [22]. 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 [16]. 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_{sn}^{-1} . Then we use the theorem of Fisher-Tippet and Gnedenko [16] to find parameters for the extreme value distribution $G(\mu, \sigma)$:

$$\begin{aligned} \mu &= \Phi_{sn}^{-1}\left(1 - \frac{1}{n}\right) \\ \sigma &= \Phi_{sn}^{-1}\left(1 - \frac{1}{n}e^{-1}\right) - \mu \end{aligned}$$

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.

5 EVALUATION

In this chapter, we present the experimental evaluation of the presented groupjoins, and the quality of our aggregate estimations in our research RDBMS Umbra [47]. 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. Afterwards, we answer the question of 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.

5.1 Groupjoin

As detailed in Section 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.

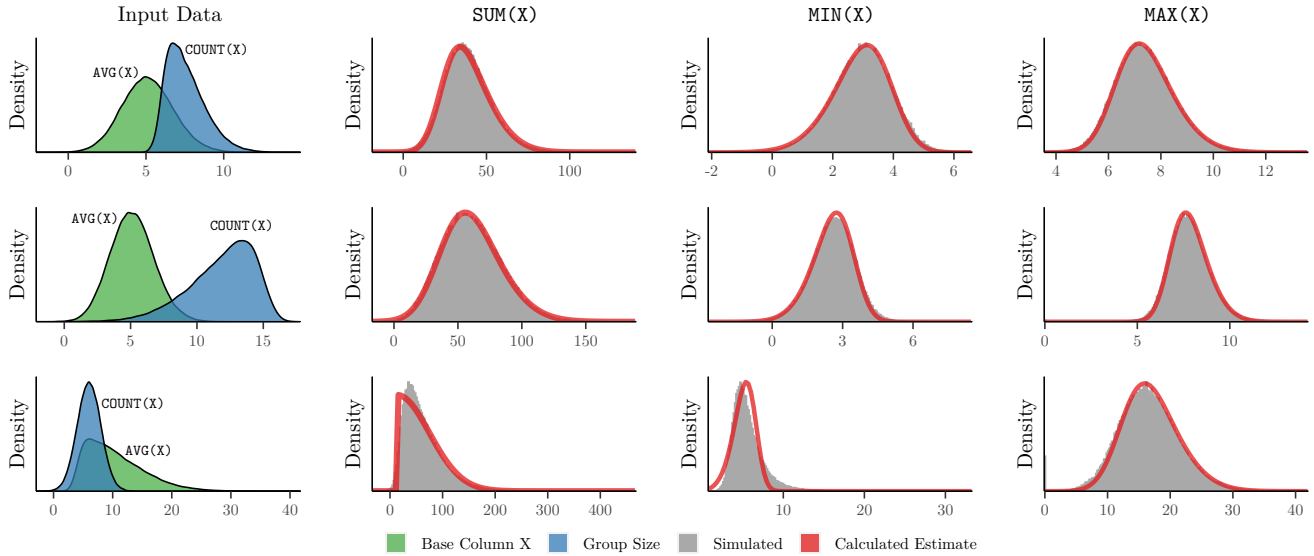


Figure 9: Skew-Normal Fit of Aggregates. The first column shows three different distributions of base column and group size. The next three columns compare simulated aggregates with a calculated fit of our skew-normal estimator.

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 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 256 GB 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 about 1%. In the first 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 second 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 [8] is about 3.2× and about 101× over PostgreSQL [57].

Cost Model We first go through the cost model calculations for groupjoins in TPC-H, before evaluating if the model accurately predicts the resulting performance. In TPC-H, we execute a total of four 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 online².

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	147 k	3.24 M	54%	6.8%	3.32 M	956 k	954 k
13	150 k	1.48 M	63%	100%	1.58 M	4.75 M	5.14 M
17	204	6.00 M	100%	0.10%	6.00 M	19.0 k	20.9 k
18	57	6.00 M	100%	0.84%	6.00 M	152 k	167 k

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, 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.

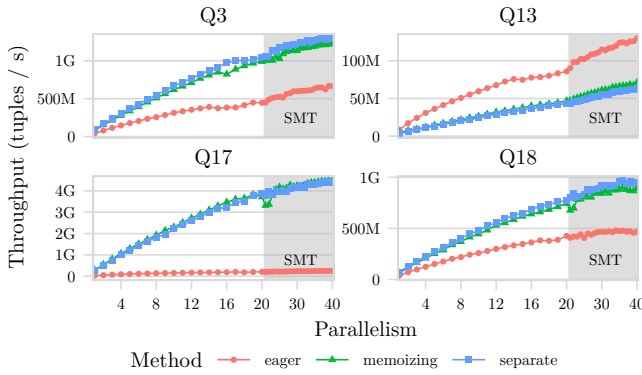


Figure 10: Parallel scale-out of TPC-H SF10 groupjoin queries.

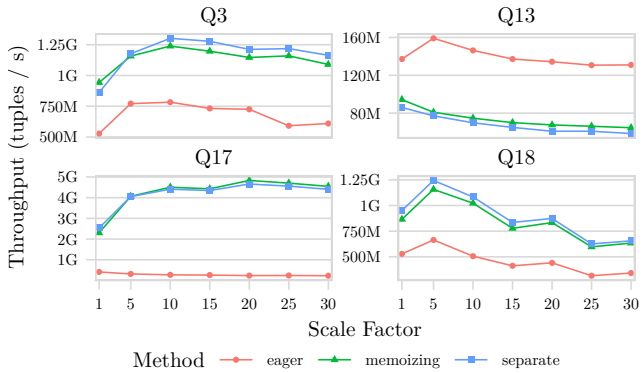


Figure 11: Data size scale-out of groupjoins in TPC-H.

In Figure 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%.

Now, we validate the quality of our cost model recommendations. This experiment compares the predicted cheapest to the actually

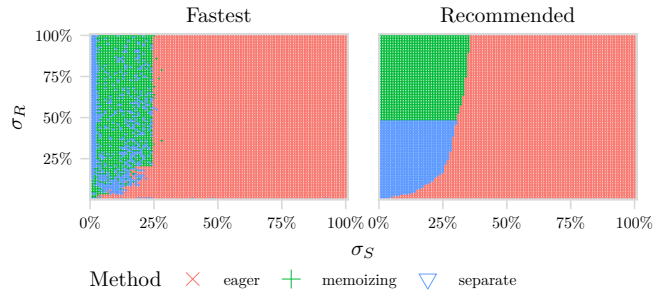


Figure 12: Comparison of fastest and cost model recommended implementation for a TPC-H orders - lineitem groupjoin.

measured fastest implementation. The setup is a micro benchmark on the TPC-H SF 10 data set with a 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 12 shows these 10 000 combinations, and plots the measured fastest implementations as points 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, if 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. The memoizing and separate executions are closer in their measured performance, since dynamic memory allocation in Umbra is very cheap. 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.

5.2 Aggregate Estimations

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 0.0.11952, DBMS X, PostgreSQL 13.1, and our research system Umbra [47]. To get accurate

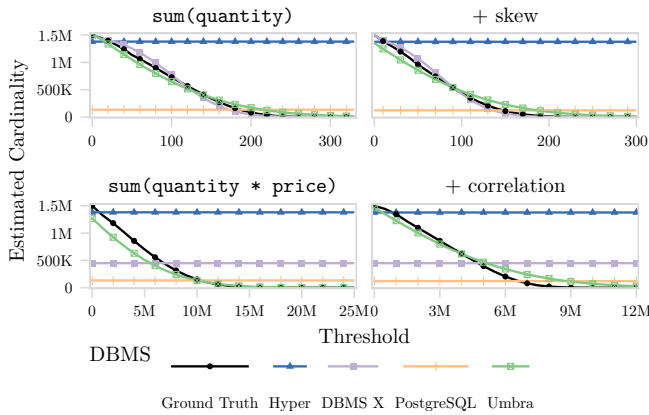


Figure 13: Estimates for TPC-H Q18 style subqueries.

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 [22].

Figure 13 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 can’t 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

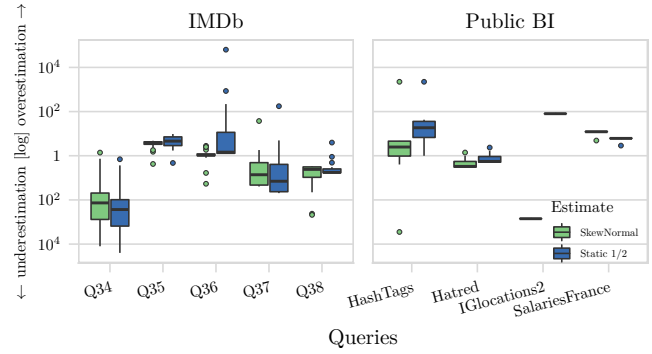


Figure 14: Estimation quality of aggregation queries. The box plots show the log-scale q-error of our estimates in comparison to the static selectivity of Hyper. Our skew-normal model reduces the geometric mean q-error by 46 % from 45.8 to 24.7.

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 consider aggregation queries derived from public workbooks in Tableau Public⁴. The query set is available online: <https://db.in.tum.de/~fent/data/aggEst.tgz>

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 [43].

In Figure 14, 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,

³<https://homepages.cwi.nl/~boncz/job/imdb.tgz>

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

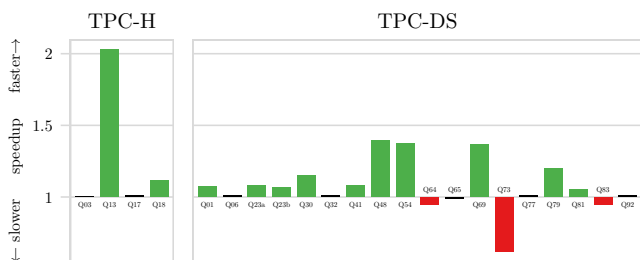


Figure 15: Overall Impact on TPC-H and TPC-DS.

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 $\frac{1}{2}$ fraction that Hyper uses, which we compare in Figure 14. 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 15 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, we are convinced that it is 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.

6 RELATED WORK

As outlined in Section 2, our work relies on well-known work on query unnesting, which enables aggregates to be embedded in the query tree [15, 21, 31, 48]. Subquery unnesting to flatten the query tree is well-known as one of the most important aspects of query optimization [7, 17]. Galindo-Legaria and Joshi [23] 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. Section 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. [4] describe the execution of correlated subqueries with window operations in Oracle. Hölsch et al. [27] 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 [20]. 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.

Big data systems for approximate query processing have proposed some approaches to answer aggregate queries [10, 13, 28, 60].

However, they use relatively large samples, which make them unsuitable for cardinality estimation, or dedicated data structures that are harder to use for ad hoc queries. A newer development is using machine learning approaches to approximately answer queries, which are cheap enough to also be used to estimate cardinalities [32, 33]. In DeepDB [25] Hilprecht et al. train Relational Sum Product Networks that learn correlations from table samples and joins, and capture the data distribution of independent subsets. This allows them to accurately estimate even correlated predicates and joins, and to calculate approximate answers to SUM, COUNT and AVG aggregates. In contrast to our work, learned approaches have significant training overhead and currently do not support MIN and MAX aggregates.


In many real-world evaluations, join and aggregation are big contributors to the overall workload [29, 58]. Consequently, there is a large body of related work that optimizes hash joins [39, 49, 55] and hash aggregations [38, 52, 61]. One often discussed question is, if hash tables should be partitioned or non-partitioned [3]. Our proposed approaches in Section 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 [14, 41]. We see this work as orthogonal, and these ideas can work hand-in-hand with parallel groupjoin execution.

7 CONCLUSION

In our paper, we presented our approach to efficiently evaluate nested aggregates in a general-purpose relational database management system. We improve two important pieces of the query engine that previously have not worked well with unnested aggregates. First, we presented a low overhead estimation of computed columns, which significantly improves the estimation quality of predicates on aggregates. The improved estimates then enable better query plans with nested aggregates. While improvements for estimates do not translate directly to improved query plans, they are an important requirement to find efficient execution 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 parallel execution of groupjoin, which commonly occur in unnested and regular aggregation queries. Our contention-free parallel groupjoin execution allows them to be more universally applicable, when they are beneficial. We also demonstrated, theoretically and practically, that using a groupjoin is not always advantageous, compared to separate join and aggregation. Our simple, yet effective cost model plans the best execution strategy, which can result in a significant speedup.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286). 

REFERENCES

- [1] Adelchi Azzalini. 1985. A class of distributions which includes the normal ones. *Scandinavian journal of statistics* (1985), 171–178.
- [2] Adelchi Azzalini. 2013. *The skew-normal and related families*. Vol. 3. Cambridge University Press.
- [3] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD Conference*. ACM, 168–180.
- [4] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zaït, and Chun Chieh Lin. 2009. Enhanced Subquery Optimizations in Oracle. *Proc. VLDB Endow.* 2, 2 (2009), 1366–1377.
- [5] Altan Birlir, Bernhard Radke, and Thomas Neumann. 2020. Concurrent online sampling for all, for free. In *DaMoN*. ACM, 5:1–5:8.
- [6] Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*. ACM, 68–78.
- [7] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC (Lecture Notes in Computer Science)*, Vol. 8391. Springer, 61–76.
- [8] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.
- [9] Damianos Chatziantoniou, Michael O. Akinde, Theodore Johnson, and Samuel Kim. 2001. The MD-join: An Operator for Complex OLAP. In *ICDE*. IEEE Computer Society, 524–533.
- [10] Surajit Chaudhuri, Gautam Das, and Vivek R. Narasayya. 2001. A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries. In *SIGMOD Conference*. ACM, 295–306.
- [11] John Cieslewicz and Kenneth A. Ross. 2007. Adaptive Aggregation on Chip Multiprocessors. In *VLDB*. ACM, 339–350.
- [12] Sophie Cluet and Guido Moerkotte. 1995. Efficient Evaluation of Aggregates on Bulk Types. In *DBPL (Electronic Workshops in Computing)*. Springer, 8.
- [13] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases* 4, 1–3 (2012), 1–294.
- [14] Andrew Crotty, Alex Galakatos, and Tim Kraska. 2020. Getting Swole: Generating Access-Aware Code with Predicate Pullups. In *ICDE*. IEEE, 1273–1284.
- [15] Umeshwar Dayal. 1987. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *VLDB*. Morgan Kaufmann, 197–208.
- [16] Laurens De Haan and Ana Ferreira. 2007. *Extreme value theory: an introduction*. Springer Science & Business Media.
- [17] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (2020), 1206–1220.
- [18] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *Proc. VLDB Endow.* 12, 9 (2019), 1044–1057.
- [19] Marius Eich, Pit Fender, and Guido Moerkotte. 2018. Efficient generation of query plans containing group-by, join, and groupjoin. *VLDB J.* 27, 5 (2018), 617–641.
- [20] Amr El-Helw, Ihab F. Ilyas, and Calisto Zuzarte. 2009. StatAdvisor: Recommending Statistical Views. *Proc. VLDB Endow.* 2, 2 (2009), 1306–1317.
- [21] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind Joshi. 2007. Execution strategies for SQL subqueries. In *SIGMOD Conference*. ACM, 993–1004.
- [22] Michael J. Freitag and Thomas Neumann. 2019. Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In *CIDR*. www.cidrdb.org.
- [23] César A. Galindo-Legaria and Milind Joshi. 2001. Orthogonal Optimization of Subqueries and Aggregation. In *SIGMOD Conference*. ACM, 571–581.
- [24] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*.
- [25] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.
- [26] Denis Hirn and Torsten Grust. 2019. PgCuckoo: Laying Plan Eggs in PostgreSQL’s Nest. In *SIGMOD Conference*. ACM, 1929–1932.
- [27] Jürgen Hölsch, Michael Grossniklaus, and Marc H. Scholl. 2016. Optimization of Nested Queries using the NF2 Algebra. In *SIGMOD Conference*. ACM, 1765–1780.
- [28] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *SIGMOD Conference*. ACM, 631–646.
- [29] Martin Kersten, Panagiotis Koutsourakis, Niels Nes, and Ying Zhan. 2021. Bridging the Chasm between Science and Reality. In *CIDR*. www.cidrdb.org.
- [30] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dube. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389.
- [31] Won Kim. 1982. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst.* 7, 3 (1982), 443–469.
- [32] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue. In *AIDB*.
- [33] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org.
- [34] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. *Proc. VLDB Endow.* 14 (2021).
- [35] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.
- [36] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.
- [37] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.
- [38] Feilong Liu, Ario Salmasi, Spyros Blanas, and Anastasios Sidiropoulos. 2018. Chasing Similarity: Distribution-aware Aggregation Scheduling. *Proc. VLDB Endow.* 12, 3 (2018), 292–306.
- [39] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2018. Many-query join: efficient shared execution of relational joins on modern hardware. *VLDB J.* 27, 5 (2018), 669–692.
- [40] Norman May and Guido Moerkotte. 2005. Main Memory Implementations for Binary Grouping. In *XSym (Lecture Notes in Computer Science)*, Vol. 3671. Springer, 162–176.
- [41] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.* 11, 1 (2017), 1–13.
- [42] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *PVLDB* 4, 11 (2011), 843–851.
- [43] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (2009), 982–993.
- [44] M. Muralikrishna. 1992. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *VLDB*. Morgan Kaufmann, 91–102.
- [45] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *VLDB*. ACM, 1049–1058.
- [46] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
- [47] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.
- [48] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW (LNI)*, Vol. P-241. GI, 383–402.
- [49] Thomas Neumann, Viktor Leis, and Alfons Kemper. 2017. The Complete Story of Joins (in HyPer). In *BTW (LNI)*, Vol. P-265. GI, 31–50.
- [50] Mark O’Neill, Elham Vaziripour, Justin Wu, and Daniel Zappala. 2016. Condensing Steam: Distilling the Diversity of Gamer Behavior. In *Internet Measurement Conference*. ACM, 81–95.
- [51] Arthur Pewsey. 2000. Problems of inference for Azzalini’s skewnormal distribution. *Journal of applied statistics* 27, 7 (2000), 859–870.
- [52] Orestis Polychroniou and Kenneth A. Ross. 2013. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*. ACM, 6.
- [53] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (2013), 1080–1091.
- [54] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *SIGMOD Conference*. ACM, 1642–1659.
- [55] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD Conference*. ACM, 1961–1976.
- [56] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1996. Cost-Based Optimization for Magic: Algebra and Implementation. In *SIGMOD Conference*. ACM Press, 435–446.
- [57] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD Conference*. ACM Press, 340–355.
- [58] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*. ACM,

1:1–1:6.

- [59] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *VLDB*. Morgan Kaufmann, 345–357.
- [60] Ying Yan, Liang Jeff Chen, and Zheng Zhang. 2014. Error-bounded Sampling for Analytics on Big Sparse Data. *Proc. VLDB Endow.* 7, 13 (2014), 1508–1519.

- [61] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. 2019. Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities. In *CIDR*. www.cidrdb.org.