Building Advanced SQL Analytics From Low-Level Plan Operators

André Kohn
Technische Universität München
andre.kohn@tum.de

Viktor Leis
Friedrich-Schiller-Universität Jena
viktor.leis@uni-jena.de

Thomas Neumann
Technische Universität München
thomas.neumann@tum.de

ABSTRACT

Analytical queries virtually always involve aggregation and statistics. SQL offers a wide range of functionalities to summarize data such as associative aggregates, distinct aggregates, ordered-set aggregates, grouping sets, and window functions. In this work, we propose a unified framework for advanced statistics that composes all flavors of complex SQL aggregates from low-level plan operators. These operators can reuse materialized intermediate results, which decouples monolithic aggregation logic and speeds up complex multi-expression queries. The contribution is therefore twofold: our framework modularizes aggregate implementations, and outperforms traditional systems whenever multiple aggregates are combined. We integrated our approach into the high-performance database system Umbra and experimentally show that we compute complex aggregates faster than the state-of-the-art HyPer system.

CCS CONCEPTS

• **Information systems → Query optimization**; **Query operators**; **Query planning**.

KEYWORDS

Database Systems; Query Optimization; Query Processing

ACM Reference Format:
André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457288

1 INTRODUCTION

Users are rarely interested in wading through large query results when extracting knowledge from a database. Summarizing data using aggregation and statistics therefore lies at the core of analytical query processing. The most basic SQL constructs for this purpose are the associative aggregation functions SUM, COUNT, AVG, MIN, and MAX, which may be qualified by the DISTINCT keyword and have been standardized by SQL-92. Given the importance of aggregation for high-performance query processing, it is not surprising that several parallel in-memory implementations of basic aggregation have been proposed [16, 22, 26, 32, 33].

While associative aggregation is common, it is also quite basic. Consequently, SQL has grown considerably since the '92 standard, obtaining more advanced statistics functionality over time. In SQL:1999, *grouping sets* were introduced, which allow summarizing data sets at multiple aggregation granularities simultaneously. SQL:2003 added *window functions*, which compute a new attribute for a tuple based on its neighboring rows. Window functions enable very versatile functionality such as ranking, time series analysis, and cumulative sums. SQL:2003 also added the capability of computing percentiles (e.g., median). PostgreSQL calls this functionality *ordered-set aggregates* because percentiles require (partially) sorted data, and we use this term throughout the paper.

Advanced statistics constructs can be used in conjunction, as the following SQL query illustrates:

```
WITH diffs AS (
  SELECT a, b, c-lag(c) OVER (ORDER BY d) AS e
  FROM R))            -- window function (lag)
SELECT a, b,
  avg(e),             -- associative aggregate
  median(e)           -- ordered-set aggregate
  count(DISTINCT e), -- distinct aggregate
FROM diffs GROUP BY (a, b)
```

The common table expression (`WITH`) computes the difference of each attribute `c` from its predecessor using the window function `lag`. For these differences, the query then computes the average, the median, and the number of distinct values.

Associative aggregates, ordered-set aggregates, and window functions not only have different syntax, but also different semantics and implementation strategies. For example, we usually prefer on-the-fly hash-based aggregation for associative aggregates but require full materialization and sorting for ordered-set aggregates and window functions. The traditional relational approach would therefore be to implement each of these operations as a separate relational operator. However, this has two major disadvantages. First, all implementations rely on similar algorithmic building blocks (such as materialization, partitioning, hashing, and sorting), which results in significant code duplication. Second, it is hard to exploit previously-materialized intermediate results. In the example query, the most efficient way to implement the counting of distinct differences may be to scan the sorted output of median rather than to create a separate hash table. An approach that computes each statistics operator separately may therefore not only require much code, but also be inefficient.

An alternative to multiple relational operators would be to implement all the statistics functionality within a single relational operator. This would mean that a large part of the query engine would be implemented in a single, complex, and large code fragment. Such an approach could theoretically avoid the code duplication and reuse problems, but we argue that it is too complex. Implementing a single efficient and scalable operator is a major undertaking [24, 26] – doing all at once seems practically impossible.

We instead propose to break up the SQL statistics functionality into several physical building blocks that are smaller than traditional relational algebra. Following Lohman [25], we call these building blocks *low-level plan operators (LOLEPOPs)*. A relational algebra operator represents a stream of tuples. A LOLEPOP, in contrast, may also represent materialized values with certain physical properties such as ordering. Like traditional operators, LOLEPOPs are composable – though LOLEPOPs often result in DAG-structured, rather than tree-structured, plans. LOLEPOPs keep the code modular and conceptually clean, while speeding up complex analytics queries with multiple expressions by exploiting physical properties of earlier computations. Another benefit of this approach is extensibility: adding a new complex statistic is straightforward.

In this paper, we present the full life cycle of a query, from translation, over optimization, to execution: In Section 3, we first describe how to translate SQL queries with complex statistical expressions to a DAG of LOLEPOPs, and then discuss optimization opportunities of this representation. Section 4 describes how LOLEPOPs are implemented, including the data structures and algorithms involved. In terms of functionality, our implementation covers aggregation in all its flavors (associative, distinct, and ordered-set), window functions, and grouping sets. As a side effect, our approach also replaces the traditional sort and temp operators since statistics operators often require materializing and sorting the input data. Therefore, this paper describes a large fraction of any query engine implementing modern SQL (the biggest exceptions are joins and set operations). We integrated the proposed approach into the high-performance compiling database system Umbra [20, 28] and compare its performance against HyPer [27]. We focus on the implementation of non-spilling LOLEPOP variants, and assume that the working-set fits into main-memory. The experimental results in Section 5 show that our system outperforms HyPer on complex statistical queries – even though HyPer has highly-optimized implementations for aggregation and window functions. We close with a discussion of related work in Section 6 and a summary of the paper in Section 7.

2 BACKGROUND

Relational algebra operators are the prevalent representation of SQL queries. In relational systems, the life cycle of a query usually begins with the translation of the SQL text into a tree-shaped plan containing logical operators such as SELECTION, JOIN, or GROUP BY. These trees of logical operators are optimized and lowered to physical operators by specifying implementations and access methods. The physical operators then serve as the driver for query execution, for example through vectorized execution or code generation. System designs differ vastly in this last execution step but usually share a very similar notion of logical operators.

Database systems usually introduce at least two different operators to support the data analysis operations of the SQL standard. The first and arguably most prominent one, is the GROUP BY operator, which computes associative aggregates like SUM, COUNT and MIN. These aggregate functions are part of the SQL:1992 standard and already introduce a major hurdle for query engines in form of the optional DISTINCT qualifier. A hash-based DISTINCT implementation will effectively introduce an additional aggregation phase that precedes the actual aggregation to make the input unique for each group. When computing the aggregate SUM(DISTINCT a) GROUP BY b, for example, many systems are actually computing:

```
SELECT sum(a)
FROM (SELECT a, b FROM R GROUP BY a, b)
GROUP BY b
```

Now consider a query that contains the two distinct aggregates SUM(DISTINCT a), SUM(DISTINCT b) as well as SUM(c). If we resort to hashing to make the attributes a and b distinct, we will receive a DAG that performs five aggregations and joins all three aggregates into unique result groups afterwards. This introduces a fair amount of complexity hidden within a single operator.

Grouping sets increase the complexity of the GROUP BY operator further as the user can now explicitly specify multiple group keys. With grouping sets, the GROUP BY operator has to replicate the data flow of aggregates for every key that the user requests. An easy way out of this dilemma is the UNION ALL operator, which allows computing the aggregates independently. This reduces the added complexity but gives away the opportunity to share results when aggregates can be re-grouped. For example, we can compute an aggregate SUM(c) that is grouped by the grouping sets (a,b) and (a) using UNION ALL as follows:

```
SELECT a, b, sum(c) FROM R GROUP BY a, b
UNION ALL
SELECT a, NULL, sum(c) FROM R GROUP BY a
```

Order-sensitive aggregation functions like median are inherently incompatible with the previously-described hash-based aggregation. They have to be evaluated by first sorting materialized values that are hash-partitioned by the group key and then computing the aggregate on the key ranges. Sorting itself, however, can be significantly more expensive than hash-based aggregation which means that the database system cannot just always fall back to sort-based aggregation for all aggregates once an order-sensitive aggregate is present. The evaluation of multiple aggregates consequently involves multiple algorithms that, in the end, have to produce merged result groups. The optimal evaluation strategy heavily depends on the function composition which presents a herculean task for a monolithic GROUP BY operator.

The WINDOW operator is the second aggregation operator that databases have to support. Unlike GROUP BY, the WINDOW operator computes aggregates in the context of individual input rows instead of the whole group. That makes a hash-based solution infeasible even for associative window aggregates. Instead, the WINDOW operator also hash-partitions the values, sorts them by partition and ordering keys, optionally builds a segment tree, and finally evaluates the aggregates for every row [24].

It is tempting to *outsource* the evaluation of order-sensitive aggregates to this second operator that already aggregates sorted values. Some database systems therefore rewrite ordered-set aggregates into a sort-based WINDOW operator followed by a hash-based GROUP BY. This reduces code duplication by delegating sort-based aggregations to a single operator but introduces unnecessary hash aggregations to produce unique result groups. For example, the

Table 1: LOLEPOPs for advanced SQL analytics. The input and output are either a tuple stream (▶, ❯) or tuple buffer (⚑, ⊐).

	Operator	In	Out	Semantics	Implementation
Transform	`PARTITION`	❯	⚑	Hash-partitions input	Materializes hash partitions (per thread), then merges across threads
	`SORT`	⊐	⚑	Sorts hash partitions	Sorts partitions with a morsel-driven variant of BlockQuicksort [14]
	`MERGE`	⊐	⚑	Merges hash partitions	Merges partitions with repeated 64-way merges
	`COMBINE`	❯	⚑	Joins unique input on the group key	Builds partitioned hash tables after materializing input. Flushes missing groups to local hash partitions and then rehashes between pipelines
	`SCAN`	⊐	▶	Scans hash partitions	Scans materialized hash partitions and indirection vectors
Compute	`WINDOW`	⊐	⚑	Aggregates windows	Evaluates multiple window frames for each row
	`ORDAGG`	⊐	▶	Aggregates sort-based	Aggregates sorted key ranges. Scans twice for nested aggregates
	`HASHAGG`	❯	▶	Aggregates hash-based	Aggregates input in fixed-size local hash tables and flushes collisions to hash partitions, then merges partial aggregates with dynamic tables
	`*`	❯	▶	Traditional operators	–

median of attribute `a` grouped by `b` can be evaluated with a pseudo aggregation function `ANY` that selects an arbitrary group element:

```
SELECT b, any(v)
FROM (SELECT b,
  median(a) OVER (PARTITION BY b) AS v FROM R)
GROUP BY b
```

We see this as an indicator that relational algebra is simply too coarse-grained for advanced analytics since it favors monolithic aggregation operators that have to shoulder too much complexity. We further believe that this is an ingrained limitation of relational algebra rooting in its reliance on (multi-)set semantics and its inability to share materialized state between operators.

3 FROM SQL TO LOLEPOPS

In this section, we first introduce the set of LOLEPOPs for advanced SQL analytics and describe how to derive them from SQL queries. We then outline selected properties of complex aggregates and how LOLEPOPs can evaluate them efficiently.

3.1 LOLEPOPs

The execution of SQL queries traditionally centers around unordered tuple streams. Relational algebra operators like joins are defined on unordered sets which allows execution engines to evaluate queries without fully materializing both inputs. State-of-the-art systems evaluate operators by processing the input tuple-by-tuple regardless of whether the execution engine implements the Volcano-style iterator model, vectorized interpretation or code generation using the push model. An exception to this rule are systems that always materialize the entire output of an operator before proceeding. This adds flexibility when manipulating the same tuples across several operators but the inherent costs of materializing all intermediate results by default is usually undesirable.

Our LOLEPOPs bridge the gap between traditional stream-based query engines and full materialization by defining operators on both unordered tuple streams and buffers. These buffers are further specified with the physical properties *partitioning* and *ordering* which allows reusing materialized tuples wherever possible. Table 1 lists eight LOLEPOPs that are both necessary and sufficient to compose advanced SQL aggregates. Of these, five *transform* materialized values and three *compute* the actual aggregates. The transform LOLEPOPs can be thought of as utility operators that prepare the input for the compute LOLEPOPs. For example, before one can compute an ordered aggregate using `ORDAGG`, the input data has to be partitioned and sorted. Buffers can be scanned multiple times, which allows decomposing complex SQL analytics into consecutive aggregations that pass materialized state between them.

For every LOLEPOP, Table 1 lists whether it produces and consumes tuple streams (▶, ❯) or tuple buffers (⚑, ⊐). These input and output types form the interface of a LOLEPOP and may be asymmetric. For example, the `PARTITION` operator consumes an unordered stream of tuples (❯) and produces a buffer that is partitioned (⚑). The `SORT` operator, on the other hand, reorders elements in place and therefore defines input and output as buffer (⊐ ⚑). Together, the two operators form a reusable building block (❯ ⚑) that materializes input values and prepares them, e.g., for ordered-set or windowed aggregation. This allows implementing complex tasks like parallel partitioned sorting in a single code fragment and enables more powerful optimizations on composed aggregates.

Most of the LOLEPOPs consume data from a single producer and provide data for arbitrary many consumers. The only exception is the operator `COMBINE` that joins multiple tuple streams on group keys. The operator differs from a traditional join in a detail that is specific to aggregation. It leverages that groups are produced at most once by every producer to simplify the join on parallel and partitioned input. We deliberately use the term `COMBINE` distinguishing the join of unique groups from generic sets.

We use dedicated LOLEPOPs to differentiate between hash-based (`HASHAGG`) and sort-based (`ORDAGG`) aggregation. Many systems implement these two flavors of aggregation by lowering a logical `GROUP BY` operator to two physical ones. However, this choice is very coarse-grained since the result is still a single physical operator that has to evaluate very different aggregates at once. With our framework, database systems can freely combine arbitrary flavors of aggregation algorithms as long as they can be defined as a LOLEPOP. Section 3.3 discusses several examples that unveil the hitherto dormant potential of such hybrid strategies. For example, while associative aggregates usually favor hash-based aggregation, we may switch to sort-aggregation in the presence of an additional ordered-set aggregate. If the required ordering is incompatible, however, it may be more efficient to combine both, hash-based and sort-based aggregation.

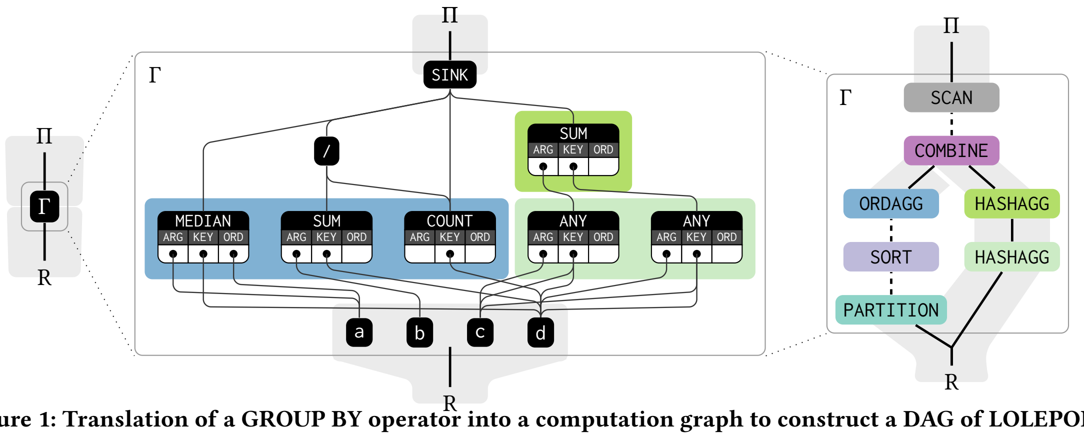


Figure 1: Translation of a GROUP BY operator into a computation graph to construct a DAG of LOLEPOPs.
SQL: `SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d`

3.2 From Tree to DAG

We derive the LOLEPOP plan from a computation graph that connects the input values, the aggregate computations as well as virtual source and sink nodes based on dependencies between them. Figure 1 shows, from left to right, a relational algebra tree, the derived computation graph, and the constructed DAG containing the LOLEPOPs for a query that computes the three aggregates median, average and distinct sum.

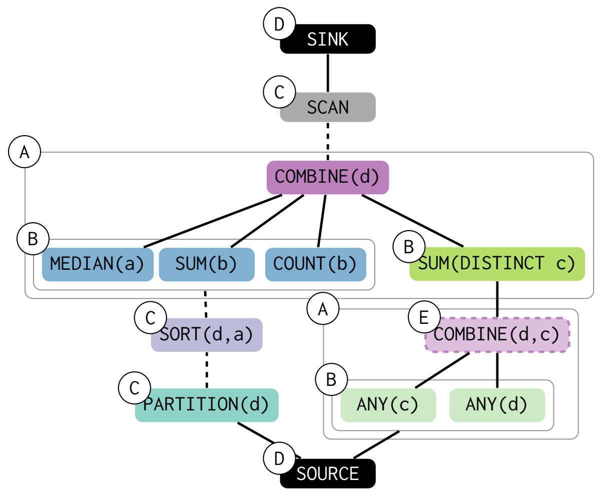
The relational algebra tree only consists of a scan, a monolithic `GROUP BY`, and a projection. At first, the `GROUP BY` aggregates are split up to unveil inherent dependencies of the different aggregation functions. The average aggregate, for example, is decomposed into the two aggregates `SUM` and `COUNT`, and a division expression. The distinct `SUM`, on the other hand, is first translated into `ANY` aggregates for arguments and keys followed by a `SUM` aggregate. `ANY` is an implementation detail that is not part of the SQL standard. It is a pseudo aggregation function that preserves an arbitrary value within a group and allows, for example, to distinguish the group keys from the unaggregated input values. Here, the attributes `c` and `d` are aggregated with `ANY`, grouped by `c`, `d` to make them unique.

The resulting aggregates and expressions are connected in the computation graph based on dependencies between them. A node in this graph depends on other nodes if they are referenced as either argument, ordering constraint or partitioning/grouping key. For example, the median computation references the attributes `a` and `d` whereas the any aggregates only depend on the attributes `c` and `d`. LOLEPOPs offer the option to compute the median independently of the distinct sum and then join the groups afterwards using the `COMBINE` operator. This results in DAG structured plans and requires special bookkeeping of the dependencies during the translation.

The computation graph of the example is translated into the seven LOLEPOPs on the right-hand side of the figure. The non-distinct aggregates in blue color are translated into the operators `PARTITION`, `SORT`, and `ORDAGG` since the median aggregate requires materializing and sorting all values anyway. The `ANY` aggregates, however, differ in the group keys and are therefore inlined as `HASHAGG` operator into the input pipeline. The distinct sum is computed in a second `HASHAGG` operator and is then joined with the non-distinct aggregates using the `COMBINE` operator.

The algorithm that derives these LOLEPOPs from the given computation graph is outlined in Figure 2. It consists of five steps that canonically map the aggregates to the LOLEPOP counterparts and then optimize the resulting DAG. Step Ⓐ collects sets of computations with similar group keys and constructs a single `COMBINE` operator for each set respectively. These `COMBINE` operators implicitly join aggregates with the same group keys and will be optimized out later if they turn out to be redundant. Step Ⓑ then constructs aggregate LOLEPOPs for computations within these sets. If the query contains grouping sets, it decomposes them into aggregations with separate grouping keys and adds them to the other aggregates attached to the combine operator. Afterwards, it divides the aggregates based on the grouping keys of their input values and determines favorable execution orders. In the example query, the operator `COMBINE(d)` joins the aggregates `MEDIAN(a)`, `SUM(b)`, `COUNT(b)` and `SUM(DISTINCT c)`. The first three aggregates depend on values that originate directly from the source operator whereas `SUM(DISTINCT c)` depends on values that are grouped by `d`, `c`. Among the first three, the aggregates `SUM` and `COUNT` are associative aggregates that would favor a hash-based aggregation. The `MEDIAN` aggregate, however, is an ordered-set aggregate and requires the input to be at least partitioned. The algorithm therefore constructs a single `ORDAGG` operator to compute the first three aggregates and a `HASHAGG` operator to compute the distinct sum. Step Ⓒ introduces all transforming operators to create, manipulate and scan buffers. In the example query, this introduces the `SORT` and `PARTITION` operators required for `ORDAGG` as well as the final `SCAN` operator to forward all aggregates to the sink. Step Ⓓ connects the source and sink operators and emits a first valid DAG.

Step Ⓔ transforms this DAG with several optimization passes. The goal of these optimization passes is to detect constellations that can be optimized and to transform the graph accordingly. In the given query, the operator `COMBINE(d,c)` can be removed since there is only a single inbound `HASHAGG` operator. Other optimizations include, for example, the merging of unbounded `WINDOW` frames into following `ORDAGG` operators if the explicit materialization of an aggregate is unnecessary or the elimination of `SORT` operators if the ordering is a prefix of an existing ordering. In addition to graph transformations, these passes are also used to configure individual operators with respect to the graph. An example is the order in which `COMBINE` operators call their producers. If, for example, a `COMBINE` operator joins two ordered-set aggregates with different ordering constraints it is usually favorable to produce



Ⓐ Add combine operators
Ⓑ Compute aggregates
- Expand grouping sets
- Select aggregation order
- Select aggregation strategies

Ⓒ Propagate buffers
- Add sorting operators
- Add partitioning operators
- Add scan operators

Ⓓ Connect DAG
- Consume from source operator
- Produce for sink operator

Ⓔ Optimize DAG
- Replace unbounded windows
- Remove redundant combines
- Select producer order
- Select buffer layouts
- Select sort modes

Figure 2: Algorithm to derive the LOLEPOP DAG.

the operator "closer" to the source first to enable in-place reordering of the buffer. In general, such a favorable producer order can be determined with a single pre-order DFS traversal starting from the plan source. Another example, is the selection of the sorting strategy in the SORT operator and the propagation of the access method to consuming operators. Very large tuples may, for example, favor indirect sorting over in-place sorting which has to be propagated to a consuming ORDAGG operator.

The result is a plan of LOLEPOPs that eliminates many of the performance pitfalls that monolithic aggregation operators will run into. We do not claim that we always find the optimal plans for the given aggregations but instead make sure that certain performance opportunities are seized. We want to explore this plan search space using a physical cost model in future research.

The algorithm translates simple standalone aggregates into chains of LOLEPOPs. An associative aggregate with distinct qualifier, for example, is translated into the sequence `HASHAGG(HASHAGG(R))`. An ordered-set aggregate is computed on sorted input using `ORDAGG(SORT(PARTITION(R)))`. For a window function, we just need to replace the last operator and evaluate `WINDOW(SORT(PARTITION(R)))` instead. This already hints at the potential code reuse between the different aggregation types but does not yet take full advantage of DAG-structured plans.

3.3 Advanced Expressions

Advanced expressions demand complex evaluation strategies. Figure 3 shows six example queries and the low-level plan.

Composed Aggregates must be split up to eliminate redundancy. The SQL standard describes various aggregation functions that can be decomposed into smaller ones. The aggregation function VAR_POP, for example, is defined as

$$Var(x) = \frac{1}{N} \cdot \sum_{i=0}^{N} (x_i - \overline{x})^2 = (\frac{1}{N} \cdot \sum_{i=0}^{N} x_i^2) - (\frac{1}{N} \cdot \sum_{i=0}^{N} x_i)^2$$

and can be decomposed into

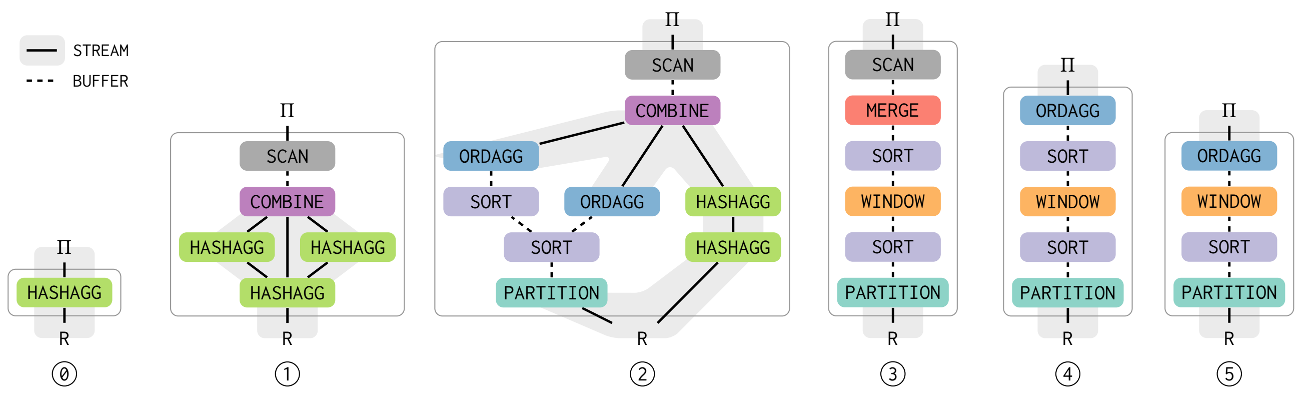
$$\frac{SUM(x^2) - \frac{SUM(x)^2}{COUNT(x)}}{COUNT(x)}.$$

We have to share the aggregate computations among and within composed aggregates, which favors a graph-like representation of aggregates and expressions. ⓪ shows a query that computes the aggregates VAR_POP(x), SUM(x), and COUNT(x). We can evaluate all three aggregates with a single hash-based aggregation operator, but still have to infer that SUM(x) and count(x) can be shared with the variance computation.

Implicit joins are necessary whenever different groups need to be served at once. This can be the result of multiple order-sensitive and distinct aggregates or an explicit grouping operation such as GROUPING SETS. ① shows a query where an associative aggregate SUM(c) is computed for the grouping sets (a), (b), and (a,b). We can evaluate the query efficiently by inlining the grouping of (a,b) into the input pipeline and then grouping (a,b) by (a) and (b). Afterwards, the output of all three aggregates is joined by (a,b) within a single hash-table. Grouping operations usually emit complicated graphs of LOLEPOPs and cause non-trivial reasoning about the evaluation order.

Order sensitivity has an invasive effect on the desirable plans since it usually requires materializing and sorting the entire input. This renders additional hash-based aggregation, which is often superior for standalone aggregation, inferior to aggregating sorted key ranges. ② shows a query that computes two order-sensitive aggregates MEDIAN(c) and MEDIAN(d) as well as two associative aggregates SUM(b) and SUM(DISTINCT b). One would usually prefer hash-bashed aggregation for the non-distinct sum aggregate to exploit the associativity. In the presence of the median aggregates, however, it is possible to compute the non-distinct sum on the same ordered key range and thus eliminate an additional hash table. The second median reuses the buffer of the first median and reorders the materialized tuples by (a,d). The distinct qualifier leaves us with the choice to either introduce two hash aggregations, grouped by (a,c) and (a), or to reorder the key ranges again by (a,c) and skip duplicates in ORDAGG. In this particular query, we use hash aggregations since the runtime is dominated by linear scans over the data as opposed to $O(n \log n)$ costs for sorting. If the key range was already sorted by (a,c), a duplicate-sensitive ORDAGG would be preferable.

Result ordering is specified through the SQL keywords ORDER BY, LIMIT and OFFSET and is crucial to reduce the cardinality of the result sets. We already rely on ordered input in the WINDOW and ORDAGG operators, which makes standalone sorting of values a byproduct of our framework. There are only two adjustments necessary. First, we have to support the propagation of LIMIT and OFFSET constraints through the DAG of LOLEPOPs to stop sorting eagerly. This can be implemented as pass through the DAG very similar to traditional optimizations of relational algebra operators. Additionally, we need a dedicated operator MERGE that uses repeated k-way merges to reduce the partition count efficiently.



```
⓪ SELECT a, var_pop(b), count(b), sum(b) FROM R GROUP BY a
① SELECT a, b, sum(c) FROM R GROUP BY GROUPING SETS ((a), (b), (a, b));
② SELECT a, sum(b), sum(DISTINCT b), percentile_disc(0.5) WITHIN GROUP (ORDER BY c),
     percentile_disc(0.5) WITHIN GROUP (ORDER BY d) FROM R GROUP BY a
③ SELECT row_number() OVER (PARTITION BY a ORDER BY b) FROM R ORDER BY c LIMIT 100
④ SELECT a, mad() WITHIN GROUP (ORDER BY b) FROM R GROUP BY a
⑤ SELECT b, sum(pow(next_a - a, 2)) / nullif(count(*) - 1, 0)
  FROM (SELECT b, a, lead(a) OVER (PARTITION BY b ORDER BY a) AS next_a FROM R GROUP BY b)
```

Figure 3: Plans for queries with aggregations that outline challenges for monolithic aggregation operators.

③ shows a query that computes the window function `row_number` of attribute `b` and then sorts the results by an attribute `c`. The traditional approach would involve a dedicated operator on top that materializes and sorts the scanned output of the window aggregation. We instead just reorder the already materialized tuples by the new order constraint and eliminate the additional materialization.

Nested aggregates blur the boundary between grouped and windowed aggregations. The *Median Absolute Deviation* (MAD), for example, is a common measure for dispersion in descriptive statistics and is defined for a set $\{x_1, x_2, ..., x_n\}$ as $MEDIAN(|x_i - \tilde{x}|)$ with $\tilde{x} = MEDIAN(x)$. $\tilde{x}$ represents a window aggregate since $x_i - \tilde{x}$ has to be evaluated for every row. The outer median, however, is an order-sensitive grouping aggregation that reduces each group to a single value. One would like to try to transform this expression into a simpler form that eliminates the nested window aggregation, similar to the aforementioned variance function. However, the nature of the median prevents these efforts and one is forced to explicitly (re-)aggregate $x_i - \tilde{x}$. ④ shows a query that computes this MAD function. Our framework allows us to first compute the window aggregate and then reorder the key ranges for a following `ORDAGG` operator. This shows the power of our unified framework, which blurs the boundary between the `GROUP BY` and `WINDOW` operators and can reuse the materialized output of $x_i - \tilde{x}$.

Nested aggregates can also be provided by the user if the database system accepts window aggregates as input arguments for aggregation functions. The *Mean Square Successive Difference* (MSSD) is defined as

$$\sqrt{\frac{\sum_{i=0}^{N-1} (x_{i+1} - x_i)^2}{n-1}}.$$

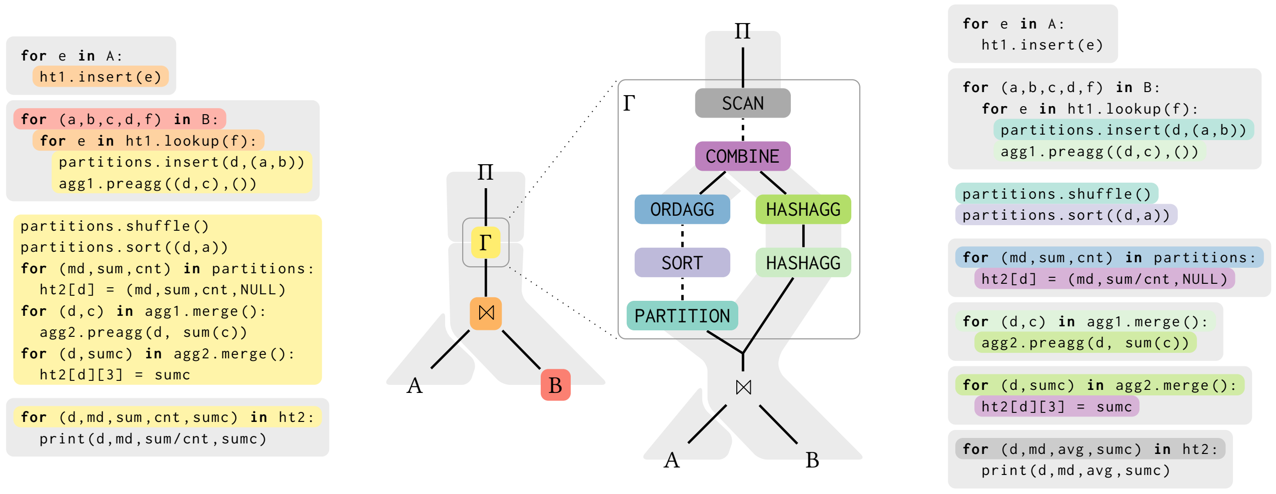
It estimates the standard deviation without temporal dispersion. ⑤ shows a query that computes the MSSD function by nesting the window aggregate `LEAD` into a `SUM` aggregate. A typical implementation would translate this query into a `WINDOW` operator followed by a `GROUP BY`. This, however, disregards the fact that the nested `WINDOW` ordering is compatible with the outer group keys. We can instead just aggregate the existing key ranges without further reordering using the `ORDAGG` operator.

3.4 Extensibility

We already mentioned a number of useful and widely-used statistics that are not part of the SQL standard, and many more exist. One advantage of our approach is that the computation graph facilitates the quick composition of new aggregation functions. We construct the computation graph using a planner API that lets us define nodes with attached ordering and key properties. We then use this API in `Low-Level-Functions` to compose complex aggregates through a sequence of API calls. In fact, we even implement the aggregation functions defined in the SQL standard as such `Low-Level-Functions`. The following example code defines the aforementioned *Mean Square Successive Difference* aggregate without explicitly implementing it in the operator logic:

```
def planMSSD(arg, key, ord):
  f = WindowFrame(Rows, CurrentRow, Following(1))
  lead = plan(LEAD, arg, key, ord, f)
  ssd = plan(power(sub(lead, arg), 2))
  sum = plan(SUM, ssd, key)
  cnt = plan(COUNT, ssd, key)
  res = plan(div(sum, nullif(sub(cnt, 1), 0)))
  return res
```

Other complex statistical functions like interquartile range, kurtosis, or central moment can be implemented similarly. Furthermore, a database system can expose this API through user-defined aggregation functions. This allows users to combine arbitrary expressions and aggregations without the explicit boundaries between the former relational algebra operators.



```
for e in A:
  ht1.insert(e)

for (a,b,c,d,f) in B:
  for e in ht1.lookup(f):
    partitions.insert(d,(a,b))
    agg1.preagg((d,c),())

partitions.shuffle()
partitions.sort((d,a))
for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum,cnt,NULL)
for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))
for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc

for (d,md,sum,cnt,sumc) in ht2:
  print(d,md,sum/cnt,sumc)
```

```
for e in A:
  ht1.insert(e)

for (a,b,c,d,f) in B:
  for e in ht1.lookup(f):
    partitions.insert(d,(a,b))
    agg1.preagg((d,c),())

partitions.shuffle()
partitions.sort((d,a))

for (md,sum,cnt) in partitions:
  ht2[d] = (md,sum/cnt,NULL)

for (d,c) in agg1.merge():
  agg2.preagg(d, sum(c))

for (d,sumc) in agg2.merge():
  ht2[d][3] = sumc

for (d,md,avg,sumc) in ht2:
  print(d,md,avg,sumc)
```

Figure 4: Plans and simplified code for a query that computes a median, an average, and a distinct sum of two joined relations.

4 LOLEPOP IMPLEMENTATION

In this section, we describe how the framework affects the code generation in our database system Umbra. We further introduce the data structures used to efficiently pass materialized values and outline the implementation of the LOLEPOPs `PARTITION` and `COMBINE` as well as the `ORDAGG`, `HASHAGG`, and `WINDOW` operators.

4.1 Code Generation

Umbra follows the producer/consumer model to generate efficient code for relational algebra plans [20, 27, 28]. In this model, operator pipelines are merged into compact loops to keep SQL values in CPU registers as long as possible. More specifically, code is generated by traversing the relational algebra tree in a depth-first fashion. By implementing the function `produce`, an operator can be instructed to recursively emit code for all child operators. The function `consume` is then used in the inverse direction to inline code of the parent operator into the loop of the child. Operators are said to *launch* pipelines by generating compact loops with inlined code of the parent operators and *break* pipelines by materializing values as necessary.

Figure 4 illustrates the code generation for a query that first joins two relations A and B and then computes the aggregates median, average, and distinct sum. The coloring indicates which line in the pseudocode was generated by which operator. On the left-hand side of the figure, the scan of the base relation B is colored in red and only generates the outermost loop of the second pipeline. The join is colored in orange and inlines code building a hash table into the first pipeline as well as code probing the hash table into the second pipeline. Both operators integrate seamlessly into the producer/consumer model since the generated loops closely match the unordered (multi-)sets at the algebra level. The group by operator, on the other hand, bypasses the model almost entirely. The code in yellow color partitions and sorts all values for the median and average aggregates and additionally computes the distinct sum via two hash aggregations. In contrast to the scan and join operators, most of this code is generated *in between* the unordered input and output pipelines since the aggregation logic primarily manipulates materialized values.

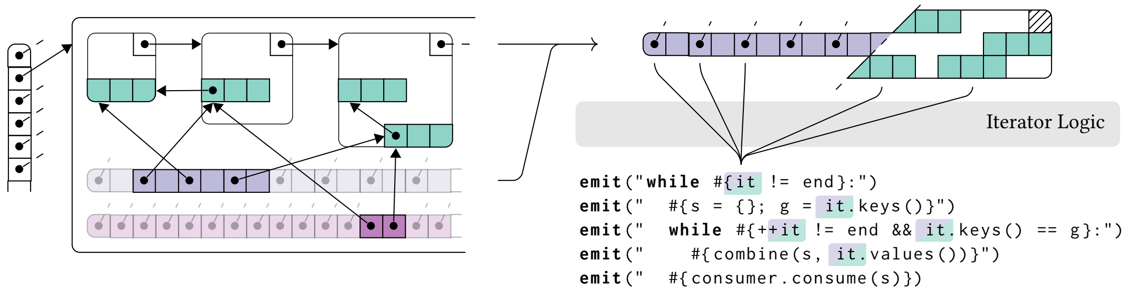
Our framework breaks this monolithic aggregation logic into a DAG of LOLEPOPs. Within the producer/consumer model, a LOLEPOP behaves just like every other operator with the single exception that it does not necessarily call consume on the parent operator. Instead, multiple LOLEPOPs can manipulate the same tuple buffer via code generated in the produce calls.

These derived DAG that roots in the two outgoing edges of the former join operator. The producer/consumer model supports DAGs since we can inline both consumers of the join (the `PARTITION` and `HASHAGG` operators) into the loop of the input pipeline. The only adjustment necessary is to substitute the total order among pipeline operators with a partial order modeling the DAG structure. Our framework further unveils pipelines that have been hidden within the monolithic translation code. The output of the `ORDAGG` and the two `HASHAGG` operators represent unordered sets that can now be defined as pipelines explicitly. The dashed edges between the operators indicate passed tuple buffers as opposed to the solid edges for pipelines. In this example, data is passed implicitly between the operators `PARTITION`, `SORT`, and `ORDAGG` through the variable `partitions`. This lifts the usual limitation to only pass tuples in generated loops and allows us to compose buffer modifications.

In the example, the code that is generated through LOLEPOPs equals the code generated by the monolithic aggregation operator. This underlines that LOLEPOPs are not fundamentally new ways to evaluate aggregates but serve as more fine-granular representation that better matches the modular nature of these functions.

4.2 Tuple Buffer

The tuple buffer is a central data structure that is passed between multiple LOLEPOPs and thereby allows reusing intermediate results. Our tuple buffer design is driven by the characteristics of code generation as well as the operations that we want to support. First, the code generated by the producer-consumer model ingests data into the buffer on a tuple-by-tuple fashion. We also do not want to rely on cardinality estimates, which are known to be inaccurate [23]. Yet, we want to avoid relocating materialized tuples whenever possible. This favors a simple list of data chunks with exponentially growing chunk sizes as the primary way to represent a buffer



```
emit("while #{it != end}:")
emit("  #{s = {}; g = it.keys()}")
emit("  while #{++it != end && it.keys() == g}:")
emit("    #{combine(s, it.values())}")
emit("  #{consumer.consume(s)}")
```

Figure 5: Tuple buffer and translator code that accesses sorted key ranges through iterator abstraction at query compile time.

partition. Second, we prefer a row-major storage layout for the tuple buffer. Our system implements a column-store for relations but materializes intermediate results as rows to simplify the generated attribute access. This will particularly benefit the SORT operator since it makes in-place sorting more cache-efficient.

However, in-place sorting also becomes inefficient with an increasing tuple size since the overhead of copying wide tuples overshadows the better cache efficiency. A common alternative is to sort a vector of pointers (or tuple identifiers) instead. These indirection vectors suffer from scattered memory accesses, but feature a constant entry size that will be beneficial once tuples get larger. This contrast leads to tradeoff between cache efficiency and robustness which oftentimes favors the latter. We instead combine the best of both worlds by introducing a third option that we call the *permutation vector*. A *permutation vector* is a sequence of entries that consist of the original tuple address followed by copied key attributes. This preserves the high efficiency of key comparisons in the operators SORT, ORDAGG, and WINDOW at the cost of a slightly more expensive vector construction.

Figure 5 shows a chunk list, a permutation vector, and a hash table of a single tuple buffer partition. The right-hand side of the figure lists an exemplary translation code for the ORDAGG operator. The code generates a loop over a sequence of tuples that aggregates key ranges and passes the results to a consumer. Keys and values are loaded through an iterator logic that abstracts the buffer access at query compile time. This way, the operator does not need to be aware of either chunks or permutation vectors but can instead rely on the iterator to emit the appropriate access code.

4.3 Aggregation

The framework uses the three aggregation operators ORDAGG, WINDOW, and HASHAGG. In Figure 5, we already outlined the ORDAGG operator, which generates compact loops over sorted tuples and computes aggregates without materializing any aggregation state. We use the ORDAGG operator whenever our input is already sorted since it spares us explicit hash tables. We further use ORDAGG to efficiently evaluate nested aggregates such as SUM(x - MEDIAN(x)). Since all values are materialized, ORDAGG can compute the nested aggregates by scanning the key range repeatedly. Traditional operators are here forced to write back the result of the median into every single row and then compute the outer aggregate using a hash join. The LOLEPOPs therefore not only spare us the hash tables but also the additional result field which will positively affect the sort performance.

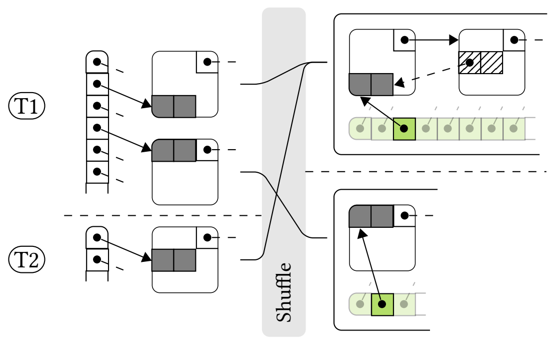


Figure 6: Two-phase hash aggregation with two threads. The hash tables on the left are fixed in size while the hash tables in green grow dynamically.

The second aggregation operator is WINDOW. Algorithmically, its implementation closely follows the window operator described by Leis et al. [24]: Within our DAG, however, the materialization, partitioning and sorting of values is delegated to other LOLEPOPs and is no longer the responsibility of the WINDOW operator. Instead, the operator begins with computing the segment trees for every hash partition in parallel. Afterwards, it evaluates the window aggregates for every row and reuses the results between window frames wherever possible. We additionally follow the simple observation that segment trees can be computed for many associative aggregates at once, independent of their frames, as long as they share the same ordering. A single WINDOW operator therefore computes multiple frames in sequence to share the segment aggregation and increase the cache-efficiency.

The third aggregation operator, HASHAGG, is illustrated in Figure 6. HASHAGG adopts a two-phase hash aggregation [22, 33]. We first consume tuples of an incoming pipeline and compute partial aggregates in fixed-size thread-local hash tables. These first hash tables use chaining, and we allocate its entries in a thread-local partitioned tuple buffer. However, we do not maintain the hash table entries in linked lists, but instead simply replace the previous entry whenever the group keys differ. This will effectively produce a sequence of partially aggregated non-unique groups that have to be merged afterwards. The efficiency of this operator roots in the ability to pre-aggregate most of its input in these local hash tables that fully reside in the CPU caches. That means that if the overall number of groups is small or if group keys are not too spread out across the relation, most of the aggregate computations will happen within this first step, which scales very well. Afterwards, the threads merge their hash partitions into a large buffer by simply concatenating the allocated chunk lists. These hash partitions are then assigned to individual threads and merged using dynamically-growing hash tables.

4.4 Partitioning

The `PARTITION` operator consumes an unordered tuple stream and produces a tuple buffer with a configurable number of, for example, 1024 hash partitions. Every single input tuple is hashed and allocated within a thread-local tuple buffer first. Once the input is exhausted, the thread-local buffers are merged across threads similar to the merging of hash partitions in the `HASHAGG` operator. Afterwards, the partition operator checks whether any of the following LOLEPOPs has requested to modify the buffer in-place. If that is the case, the partition operator compacts the chunk lists into a single chunk per partition. We deliberately introduced this additional compaction step to simplify the buffer modifications. The alternative would have been to implement all in-place modifications in a way that is aware of chunk-lists. This is particularly tedious for algorithms like sorting and also makes the generated iterator arithmetic in operators like `WINDOW` and `ORDAGG` more expensive.

4.5 Combine

The `COMBINE` operator joins unique groups on their group keys. Consider, for example, a query that pairs a distinct with a non-distinct aggregate. Both aggregates are computed in different `HASHAGG` LOLEPOPs but need to be served as single result group. The `COMBINE` operator joins an arbitrary number of input streams with the assumption that these groups are unqiue. We simply check for every incoming tuple whether a group with the given key already exists. If it is, we can just update the group with the new values and proceed. Otherwise, we materialize the group into a thread-local tuple buffer. After every pipeline, the `COMBINE` operator merges these local buffers and rehashes the partitions if necessary.

5 EVALUATION

In this section, we experimentally evaluate the planning and execution of advanced SQL analytics in Umbra using LOLEPOPs. We first compare the execution times of advanced analytical queries with Hyper, a database system that implements aggregation using traditional relational algebra operators. We then analyze the impact of aggregates on five TPC-H queries with a varying number of joins. Additionally, we show the performance characteristics of certain LOLEPOPs based on four execution traces. Our experiments have been performed on an Intel Core i9-7900X with 10 cores, 128 GB of main memory, LLVM 9, and Linux 5.3.

5.1 Comparison with other Systems

We designed a set of queries to demonstrate the advantages of our framework over monolithic aggregation operators. We chose the main-memory database system HyPer as reference implementation for *traditional* aggregation operators since it also employs code generation with the LLVM framework for best-of-breed performance in analytical workloads. The design of Umbra shares many similarities with HyPer besides the query engine which allows for a fair comparison of the execution times. Both systems rely on Morsel-Driven Parallelization [22] and compile queries using LLVM [27].

Table 2: Execution times in seconds of queries with simple aggregates in HyPer, PostgreSQL and MonetDB.

Query	HyPer	PgSQL	MonetDB
`SUM(q) GROUP BY k`	0.50	4.03	0.64
`SUM(q) GROUP BY ((k,n),(k))`	0.55	42.31	4.77
`PCTL(q,0.5) GROUP BY k`	0.89	32.96	10.19
`ROW_NUMBER() PARTITION BY k ORDER BY q`	0.87	26.58	10.36

`n=linenumber   q=quantity   k=suppkey`

The database systems PostgreSQL and MonetDB were excluded due to their lacking performance for basic aggregates. The following table compares the execution times between HyPer, PostgreSQL and MonetDB for an associative aggregate, an ordered-set aggregate and a window function as well as for grouping sets with two group keys. Our queries represent complex and composed versions of these aggregates and will increase the margin between the systems further.

The experiment differs from benchmarks such as TPC-H or TPC-DS in that the queries are not directly modeling a real-world interaction with the database. We instead define queries that only aggregate a single base table without further join processing. We focus on the relation *lineitem* of the benchmark TPC-H since it is well-understood and may serve as placeholder for whatever analytical query precedes. This does not curtail our evaluation since those operators usually form the very top of query plans and any selective join would only reduce the pressure on the aggregation logic. Our performance evaluation comprises 18 queries across five different categories. Table 3 shows the execution times using Umbra and HyPer with 1 and 20 threads and the factors between them.

Queries 1, 2, and 3 provide descriptive statistics for the single attribute `extendedprice` with a varying number of aggregation functions. The aggregates in all three queries have to be optimized as a whole since they either share computations or favor different evaluation strategies. Query 2 presents a particular challenge for monolithic aggregation operators since the function `percentile (PCTL)` is not associative. The associative aggregates `SUM`, `COUNT`, and `VAR_SAMP` can be computed on unordered streams and can be aggregated eagerly in thread-local hash tables. Non-associative aggregates like `PCTL`, on the other hand, require materialized input that is sorted by at least the group key. HyPer delegates this computation to the `Window` operator and computes the associative aggregates using a subsequent hash-based grouping. Umbra computes all aggregates on the sorted key range using the `ORDAGG` operator, which spares us the hash tables.

Queries 4, 5, 6, and 7 target the scalability of ordered-set aggregates. All four queries are dominated by the time it takes to sort the materialized values and therefore punish any unnecessary reorderings. The databases have to optimize the plan with respect to the ordering constraints to eliminate redundant work in query 5 and 6. Query 7 additionally groups by the attribute `linenumber` which contains only seven distinct values across the relation. HyPer

Table 3: Execution times in seconds for advanced SQL queries on the TPC-H lineitem table (scale factor 10).

				1 thread			20 threads		
	#	Aggregates		Umbra	HyPer	×	Umbra	HyPer	×
Single	1	`SUM(e), COUNT(e), VAR_SAMP(e)`	`GROUP BY k`	3.10	4.73	1.53	0.37	0.60	1.62
	2	`↳, PCTL(e,0.5)`	`GROUP BY k`	4.32	9.36	2.17	0.47	0.96	2.03
	3	`COUNT(e), COUNT(DISTINCT e)`	`GROUP BY k`	9.61	127.63	13.28	1.21	26.52	21.90
Ordered-Set	4	`PCTL(e,0.5)`	`GROUP BY k`	4.00	8.88	2.22	0.43	0.92	2.14
	5	`↳, PCTL(e,0.99)`	`GROUP BY k`	4.02	12.66	3.15	0.42	1.40	3.31
	6	`↳, PCTL(q,0.5), PCTL(q,0.9)`	`GROUP BY k`	6.48	22.39	3.46	0.64	2.68	4.20
	7	`PCTL(e,0.5), PCTL(q,0.5)`	`GROUP BY n`	6.74	21.93	3.25	0.93	19.85	21.36
Grouping-Sets	8	`SUM(q)`	`GROUP BY ((k,n),(k),(n))`	2.30	10.73	4.66	0.28	1.09	3.96
	9	`SUM(q)`	`GROUP BY ((k,s,n),(k,s),(k,n),(n))`	2.63	16.37	6.22	0.42	1.71	4.09
	10	`PCTL(q,0.5)`	`GROUP BY ((k,n),(k))`	2.43	18.11	7.46	0.24	1.85	7.56
	11	`PCTL(q,0.5)`	`GROUP BY ((k,s,n),(k,s),(k))`	2.77	27.78	10.05	0.31	2.89	9.44
	12	`PCTL(q,0.5)`	`GROUP BY ((k,n),(k),(n))`	1.97	26.60	13.50	0.52	10.43	20.20
Window	13	`LEAD(q), LAG(q)`	`PARTITION BY k ORDER BY r`	8.33	13.69	1.64	0.97	1.46	1.50
	14	`↳, CUMSUM(q)`	`PARTITION BY k ORDER BY d`	12.77	19.05	1.49	1.56	2.27	1.46
	15	`CUMSUM(q)`	`PARTITION BY n ORDER BY d`	5.10	12.32	2.42	0.89	10.93	12.29
Nested	16	`PCTL(e - PCTL(e,0.5),0.5)`	`GROUP BY k`	6.35	12.39	1.95	0.69	1.44	2.07
	17	`PCTL(SUM(q), 0.5)`	`GROUP BY k`	1.58	4.08	2.58	0.20	0.52	2.62
	18	`SUM(POW(LEAD(q) - q,2)) / COUNT(*)`	`GROUP BY k`	5.63	10.90	1.94	0.58	1.09	1.89

`e=extendedprice`	`n=linenumber`	`s=linestatus`	`o=orderkey`	`p=partkey`
`q=quantity`	`r=receiptdate`	`k=suppkey`	`d=shipdate`	`m=shipmode`

does not to sort partitions in parallel and is therefore considerably slower when scaling to 20 threads.

Queries 8, 9, 10, 11, and 12 analyze grouping sets that introduce a significant complexity in the aggregation logic by combining different group keys. This offers potential performance gains through reaggregation of associative aggregates and stresses the importance of optimized sort orders. HyPer only supports grouping sets by computing the different groups independently and combining the results using `UNION ALL`. With LOLEPOPs, we instead start grouping by the longest group keys first and then reaggregate key prefixes whenever necessary. In query 8, for example, we first group by (`suppkey`, `linenumber`) and then reaggregate the results by `suppkey` afterwards. Queries 10, 11, and 12 use the `percentile` function and emphasize the sort optimizations of our framework. We compute the queries 10 and 11 efficiently on a single buffer that is partitioned by attribute `suppkey`. We reorder the buffer by the constraints arranged in decreasing lengths, i.e., (`suppkey`, `linenumber`, `quantity`) followed by (`suppkey`, `quantity`) for query 10. Query 12 adds (`linenumber`) as additional group key which will again penalize systems that sort key ranges in a single-threaded fashion.

Queries 13, 14, and 15 target the scaling (in terms of number of expressions) of window queries. Query 13 combines the two window functions `LEAD` and `LAG` that can be evaluated on the same key ranges. Query 14 adds a cumulative sum on a different ordering attribute which favors an efficient reordering of the previous key range. Query 15 partitions by `linenumber` again to underline the importance of parallel sorting for all flavors of ordered aggregation.

Queries 16, 17, and 18 compose complex aggregates from window and grouping aggregates. Query 16 computes the Median Absolute Deviation (MAD) function that we described as advanced aggregate in Section 3.3. It first computes a median m of the attribute `extendedprice` as window aggregate and then reorders the buffer to compute the median of (`extendedprice`$-m$) as ordered-set aggregate. With LOLEPOPs, we can explicitly reorder the partitioned buffer by the first computed median aggregate and then compute the second median with a `ORDAGG` operator. Query 18 computes the also aforementioned function Mean Square Successive Difference (MSSD) that sums up the square difference between the window function `LEAD` and a value. This time, we do not need to reorder values but can directly compute the result on the sorted key range using `ORDAGG`. They query also shows that the performance of *traditional* aggregation operators is sometimes saved by coincidence due to almost-sorted tuple streams. In HyPer, the `WINDOW` operator streams the key ranges to the hashing `GROUP BY` almost in order, improving the effectiveness of thread-local pre-aggregation.

In summary, these 18 queries show scenarios that occur in real-world workloads and already profit from the optimizations on a DAG of LOLEPOPs. These optimizations are quite difficult to implement in relational algebra, but can be broken up into composable blocks with LOLEPOPs.

5.2 Advanced Aggregates in TPC-H

We next analyze the performance impact of advanced aggregates on TPC-H. Figure 7 shows the execution times of the TPC-H queries 4, 5, 7, 10, and 12 with and without additional aggregates at scale factor 10. The modifications of the individual queries only consist of up to two additional ordered set aggregates with different orderings or a prefix of the group key as additional grouping set.

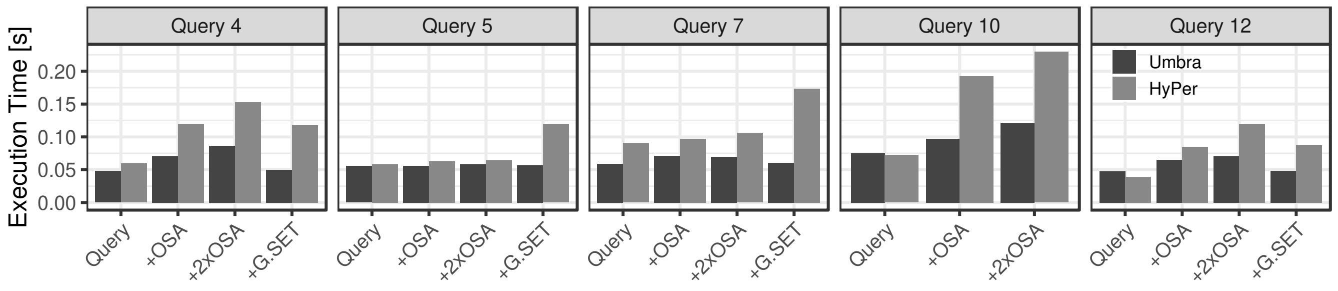


Figure 7: Execution times of five TPC-H queries at scale factor 10 with and without additional aggregates.

Query 5 and 7 contain five joins that pass only few tuples to the topmost GROUP BY operator. Both queries are dominated by join processing and the additional ordered-set aggregates, percentiles of l_quantity and l_discount, have an insignificant effect on the overall execution times. Yet, the additional grouping by either n_name or l_year doubles the execution times in HyPer since the joins are duplicated using UNION ALL. This suggests that, even without Low-Level-Plan operators, a system should at least introduce temporary tables to share the materialized join output between different GROUP BY operators.

Query 4 only contains a single semi join that filters 500,000 tuples of the relation orders. This increases the pressure on the aggregation logic which results in a slightly faster execution with Low-Level-Plan operators. These differences are further pronounced in the modified queries computing additional percentiles of o_totalprice and o_shippriority and grouping by o_orderstatus. Query 12 behaves very similar to query 4 and aggregates 300,000 tuples. HyPer is slightly faster when computing the original aggregates but loses when adding the percentiles l_quantity and l_discount or grouping by l_linestatus.

Query 10 aggregates over one million tuples produced by three joins and is also slightly faster in HyPer. However, Umbra outperforms HyPer by a factor of almost two when additionally computing the percentiles l_quantity and l_discount. This is attributable to the high number of large groups that are accumulated by the aggregation operator. The aggregation yields over 300,000 groups that are reduced with a following top-k filter. As a result, traditional hash-based aggregation suffers from the cache-inefficiency of larger hash tables. Umbra loses slightly against HyPer when computing the single sum but wins as soon as the ordered-set aggregates can eliminate the hash aggregation entirely.

The experiment demonstrates that minor additions to the well-known TPC-H queries such as adding a single ordered-set aggregate or appending a grouping set suffice to unveil the inefficiencies of monolithic aggregation operators. It also shows that queries may very well be dominated by joins, leaving only insignificant work for a final summarizing aggregation operator. In such cases, the efficiency of the aggregation is almost irrelevant which is not changed by introducing LOLEPOPs.

5.3 LOLEPOPs in Action

In a next step, we illustrate the different performance characteristics of LOLEPOPs based on two different example queries. Both

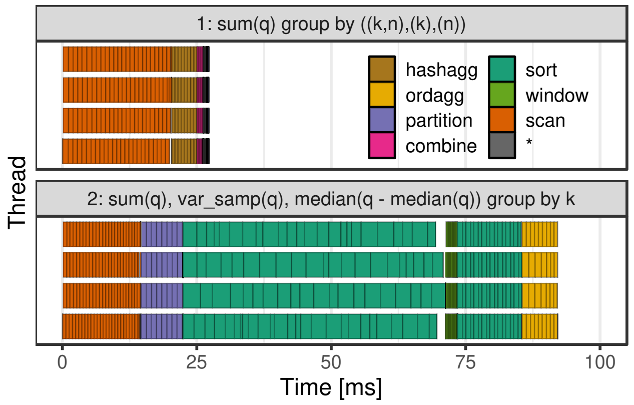


Figure 8: Execution traces of two queries on the TPC-H schema at scale factor 0.5 with four threads and 16 buffer partitions.

queries target the relation lineitem of TPC-H. We execute the queries at scale factor 0.5 with four threads and reduce the number of partitions in tuple buffers to 16 to make morsels graphically distinguishable. Figure 8 shows precise timing information about the morsels being processed.

Query 1 computes the aggregate SUM grouped by the grouping sets (suppkey, linenumber), (suppkey), and (linenumber). It is significantly faster than the second query although it groups the input using three different HASHAGG operators. Umbra computes these grouping sets efficiently by pre-aggregating the 3 million tuples of the $\boxed{\text{first}}$ scan pipeline by (suppkey, linenumber). The $\boxed{\text{second}}$ pipeline then merges these partial aggregates into 35,000 groups using dynamic hash tables for each of the 16 partitions. The $\boxed{\text{third}}$ pipeline scans the results afterwards, re-aggregates them by suppkey and linenumber, and passes them to the COMBINE operator. All remaining pipelines are barely visible since they operate on only a few tuples. The plan of this query corresponds to query ① in Figure 3.

Query 2 computes the associative aggregates SUM and VAR_SAMP, as well as the Median Absolute Deviation that was introduced as advanced aggregate in Section 3. We include it as execution trace in this experiment to illustrate the advantages of sharing materialized state between operators. Umbra evaluates this query by computing the nested median as window expression and then reordering the results in place. In contrast to the previous query, the first pipeline is now faster since it only materializes the tuples into 16 hash partitions. Thereafter, the compaction merges the chunk list of each hash partition into single chunks that enable the in-place modifications. The fourth pipeline represents the window operator that computes the median for every key range and stores the result in every row. The following pipeline then reorders the partitioned buffer by this median value. It is significantly faster than the first sort since the hash partitions are already sorted by the key. The last pipeline then iterates over the sorted key-ranges and computes the three aggregates at once.

6 RELATED WORK

Research on optimizing in-memory query processing on modern hardware has been growing in the past decade. However, in comparison to joins [1–7, 17, 21, 34], work on advanced statistics operators is relatively sparse. Efficient aggregation is described by a number of papers [16, 22, 26, 32, 33]. However, these papers omit to discuss how to implement DISTINCT aggregates, which are significantly more complicated than simple aggregates, in particular, when implemented using hashing. There is even less work on window functions: The papers of Leis et al. [24] and Wesley et al. [36] are the only one that describe the implementation of window functions in detail. Cao et al. [9] present query optimization techniques for queries with multiple window functions (e.g., reusing existing partitioning and ordering properties), which are also applicable and indeed are directly enabled by our approach. Except for Xu et al. [37], much work on optimizing sort algorithms for modern hardware [10, 14] has focused on small tuple sizes. Grouping sets have been proposed by Gray et al. [19] in 1997, and consequently there have been many proposals for optimizing the grouping order: Phan and Michiardi's [30] fairly recent paper offers a good overview. We are not aware of any research papers describing how to efficiently implement ordered-set aggregates in database systems. Even more importantly, all these papers focus on a small subset of the available SQL functionality and do not discuss how to efficiently execute queries that contain multiple statistical expressions like the introductory example in Section 1. Given the importance of statistics for data analytics, this paper fills this gap by presenting a unified framework that relies on many of the implementation techniques found in the literature.

Our approach uses the notion of LOw-LEvel Plan OPerators (LOLEPOP), which was proposed in 1988 by Lohman [25] and is itself based on work by Freytag [18]. According to them, a LOLEPOP may either be a traditional relational operator (e.g., join or union) or a low-level building block (e.g., sort or ship). The result of a LOLEPOP may either be a buffer or a stream of tuples. Furthermore, a LOLEPOP may have *physical properties* such as an ordering. A concept similar to LOLEPOPs was very recently described by Dittrich and Nix [13], who focus on low-level query optimizations. They introduce the concept of Materialized Algorithmic Views (MAVs), that represent materialized results at various granularity levels in the query plan. We understand our LOLEPOPs to be an instantiation of MAVs at a granularity that is slightly lower than relational algebra. Our framework further represents aggregates as directed acyclic graphs and therefore might benefit from existing research on parallel dataflows [15]. There are also some similarities with low-level algebras [29, 31] and compilation frameworks [8, 11, 12]. However, these papers generally focus on simple select-project-join queries and on portability across heterogeneous hardware, but do not discuss how to translate and execute complex statistical queries.

The optimization of query plans with respect to *interesting sort orders* dates back to a pioneering work of Selinger et al. [35]. They consider a sort order *interesting* if it occurs in `GROUP BY` or `ORDER BY` clauses or if it is favored by join columns. These sort orders are then included during access path selection, for example, to introduce *merge joins* for orders that are required anyway. AWS Redshift is a distributed commercial system that uses *interesting* orders to break up certain aggregations at the level of relational algebra operators. Redshift introduces the operator *GroupAggregate* that consumes materialized tuples from a preceding *Sort* operator to compute ordered-set aggregates more efficiently. Our system generalizes this idea and performs access path selection with *interesting* orderings and partitionings to derive LOLEPOPs for all kinds of complex aggregation functions.

7 SUMMARY AND FUTURE WORK

The SQL standard offers a wide variety of statistical functionalities, including associative aggregates, distinct aggregates, ordered-set aggregates, grouping sets, and window functions. In this paper we argue that relational algebra operators are not well suited for expressing complex SQL queries with multiple statistical expressions. Decomposing complex expressions into independent relational operators may lead to sub-optimal performance because query execution is generally derived directly from this execution plan. We instead propose a set of low-level plan operators (LOLEPOPs) for SQL-style statistical expressions. LOLEPOPs can access and transform buffered intermediate results and, thus, allow reusing computations and physical data structures in a principled fashion. Our framework subsumes the sort-based and hash-based aggregation as well as several non-statistical SQL constructs like ORDER BY and WITH. We presented our LOLEPOP implementations and integrated our approach into the high-performance database system Umbra. The experimental comparison against the HyPer shows that LOLEPOPs improve the performance of queries with complex aggregates substantially. However, the proposed building blocks are also applicable to other query engine types, for example to vectorized engines. We also believe that our approach leads to more modular and maintainable code. The paper describes a canonical translation based on heuristic optimization rules. It would be interesting to investigate cost-based optimization strategies to further improve the plan quality. Our system Umbra further assumes that the working set fits into main memory which may no longer hold when scaling to multiple machines in the cloud. We want to explore the evaluation of advanced aggregates in a distributed setting and with constrained memory sizes, for example by dynamically switching between spilling and non-spilling LOLEPOP variants.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. 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] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB* 5, 10 (2012).

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB* 7, 1 (2013).

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*.

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015).

[5] 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*.

[6] Ronald Barber, Guy M. Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi K. Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-Efficient Hash Joins. *PVLDB* 8, 4 (2014).

[7] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*.

[8] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *VLDB J.* 27, 6 (2018).

[9] Yu Cao, Chee-Yong Chan, Jie Li, and Kian-Lee Tan. 2012. Optimization of Analytic Window Functions. *PVLDB* 5, 11 (2012).

[10] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB* 1, 2 (2008).

[11] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *PVLDB* 8, 12 (2015).

[12] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2015. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*.

[13] Jens Dittrich and Joris Nix. 2020. The Case for Deep Query Optimisation. In *CIDR*.

[14] Stefan Edelkamp and Armin Weiß. 2019. BlockQuicksort: Avoiding Branch Mispredictions in Quicksort. *ACM J. Exp. Algorithmics* 24, 1 (2019).

[15] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. 2012. Spinning Fast Iterative Data Flows. *Proc. VLDB Endow.* 5, 11 (2012).

[16] Ziqiang Feng and Eric Lo. 2015. Accelerating aggregation using intra-cycle parallelism. In *ICDE*.

[17] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *PVLDB* 13, 11 (2020).

[18] Johann Christoph Freytag. 1987. A Rule-Based View of Query Optimization. In *SIGMOD*.

[19] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.* 1, 1 (1997).

[20] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *VLDB J.* 30 (2021).

[21] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively Parallel NUMA-aware Hash Joins. In *IMDM*.

[22] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*.

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015).

[24] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *PVLDB* 8, 10 (2015).

[25] Guy M. Lohman. 1988. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *SIGMOD*.

[26] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *SIGMOD*.

[27] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011).

[28] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[29] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High Performance Data Analysis. In *CIDR*.

[30] Duy-Hung Phan and Pietro Michiardi. 2016. A novel, low-latency algorithm for multiple Group-By query optimization. In *ICDE*.

[31] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB* 9, 14 (2016).

[32] Orestis Polychroniou and Kenneth A. Ross. 2013. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*.

[33] 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. *PVLDB* 6, 11 (2013).

[34] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*.

[35] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/582095.582099

[36] Richard Wesley and Fei Xu. 2016. Incremental Computation of Common Windowed Holistic Aggregates. *PVLDB* 9, 12 (2016).

[37] Wenjian Xu, Ziqiang Feng, and Eric Lo. 2016. Fast Multi-Column Sorting in Main-Memory Column-Stores. In *SIGMOD*.