Efficient Evaluation of Arbitrarily-Framed Holistic SQL Aggregates and Window Functions

Adrian Vogelsgesang
salesforce.com, Inc.
avogelsgesang@salesforce.com

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

Viktor Leis
Friedrich-Alexander-Universität Erlangen-Nürnberg
viktor.leis@fau.de

Alfons Kemper
Technische Universität München
kemper@in.tum.de

ABSTRACT

Window functions became part of the SQL standard in SQL:2003 and are widely used for data analytics: Percentiles, rankings, moving averages, running sums and local maxima are all expressed as window functions in SQL. Yet, the features offered by SQL's window functions lack composability: Framing is only available for distributive and algebraic aggregate functions, but not for holistic aggregates like percentiles and window functions like ranks. The SQL standard explicitly disallows holistic aggregates from being framed and thereby severely limits data analysts.

This paper proposes to remove this restriction, thereby making window functions fully composable. The newly gained composability allows for more complex aggregates which are tricky to evaluate. The lack of subquadratic, parallel algorithms to evaluate framed holistic aggregates is probably *the* main objection against adding truly composable window functionality to the SQL standard. As such, this paper shows how to efficiently evaluate all window and aggregate functions from SQL:2011, except for DENSE_RANK, in combination with arbitrary window frames. This includes framed distinct aggregates, framed value functions, framed percentiles and framed ranks.

CCS CONCEPTS

• **Information systems** → *Online analytical processing engines.*

KEYWORDS

Database Systems; Holistic aggregates; Window functions

ACM Reference Format:
Adrian Vogelsgesang, Thomas Neumann, Viktor Leis, and Alfons Kemper. 2022. Efficient Evaluation of Arbitrarily-Framed Holistic SQL Aggregates and Window Functions. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526184

1 INTRODUCTION

Window functions efficiently answer important business questions involving rankings and percentiles. As such, they are commonly supported by both research and commercial systems, including Hyper [27], Umbra [24], DuckDB [2], SQLite [5], PostgreSQL [3], Snowflake [21], Oracle [29], and Microsoft SQL Server [28].

Common business questions such as "How many monthly-active users do we have?", "How long do my customers need to wait for their orders?", and "What is the 99th percentile worst-case delivery time of a product?" can be answered through window functions, by using distinct counts, ranking, or percentile functions. However, the immediate follow-up question tends to be "How did those numbers change over time? Are we getting better or worse?". Usually, one would query for such a change over time using SQL's windowing functionality, more specifically using a window frame. One could try to query for the change in monthly-active users with the query

```
select o_orderdate, count(distinct o_custkey) over w
from orders
window w as (
  order by o_orderdate
  range between '1 month' preceding and current row)
```

However, SQL:2011 explicitly disallows the usage of distinct aggregates as window functions. Similarly, the change over time in the 99th percentile of delivery times could be queried with

```
select l_shipdate,
  percentile_disc(
    0.99, order by l_receiptdate - l_shipdate
  ) over w
from lineitem
window w as (
  order by l_shipdate
  range between '1 week' preceding and current row)
```

But, again, SQL:2011 does not provide such flexibility.

Given the importance of the underlying business question, it is of little surprise that the data exploration tool *Tableau* supports moving percentiles [32] through an implementation in the application layer. It would be preferable to push down this computation to the database and profit from the composability with the rest of SQL. The Tableau researchers Wesley and Xu previously explored incremental algorithms for holistic aggregates [38]. Implementing Tableau's algorithms in a database system poses multiple challenges, though. The algorithms are single-threaded and cannot be parallelized. Some of their algorithms have a complexity of $O(n^2)$. Thereby, those algorithms do not scale to the large data sizes common in database systems. At the same time, they only present algorithms for distinct counts, modes and moving percentiles but leaves out other window functions, in particular ranking functions.

The main contributions of this paper are algorithms for holistic aggregates which

(1) cover all SQL aggregate and window functions except for `DENSE_RANK`,
(2) have a guaranteed worst-case runtime of $O(n \log n)$,
(3) can be efficiently parallelized,
(4) can reuse large parts of existing algorithms already present in most database systems (thereby reducing implementation effort), and
(5) beat state-of-the art algorithms in a performance evaluation.

Given our contributions, we argue that the existing restrictions in SQL:2011 are unnecessary.

2 WINDOW FUNCTIONS IN SQL

Aggregate functions and window functions are at the heart of almost every OLAP query [22, 37]: While most other SQL operators process each tuple in isolation, aggregation and window functions allow analysis across multiple tuples. Aggregate functions summarize multiple rows into a single row, producing one row per group. They reduce the number of tuples. In contrast, window functions do not change the number of tuples. Instead, they add additional context to each row in the form of a new column. Depending on the used window function, this new column can represent, e.g., a previous value, a rank, a running sum or a moving average.

In this section we first give an overview of aggregate functions and window functions. Based on that understanding, we then point out existing restrictions and propose solutions to fill those gaps.

2.1 Aggregate Functions

Aggregate functions summarize input tuples into an aggregate value. They are commonly divided into three complexity classes [20]:

- For *distributive* aggregates (`COUNT`, `SUM`, ...) one can combine multiple partial results to form the final result.
- *Algebraic* aggregates (`AVG`, ...) can be decomposed into *distributive* aggregates. E.g., `AVG(x) = SUM(x) / COUNT(x)`.
- *Holistic* aggregates (`PERCENTILE`, `COUNT DISTINCT`, ...) are aggregates which need to look at all input tuples holistically.

Distributive and algebraic aggregates are cheap to evaluate: While iterating over the input tuples, tuples can be eagerly combined into the aggregation state. The required memory is constant and the execution time is linear in the input size. In contrast, evaluating a holistic aggregate is more involved: A percentile cannot be computed by combining the percentiles from subsets of the input. The tuples cannot be pre-aggregated and the complete input must be inspected *holistically* to determine the resulting percentile.

2.2 Window Frames in SQL

Windowing in SQL is expressed with the syntax

```
function(arguments) over (partition by ...
  order by ... rows between ...)
```

The `OVER` clause specifies which other tuples to consider when evaluating the window function for a given row. The function call then condenses those tuples into the value for the new column.

The `OVER` clause consists of three optional parts: The `PARTITION BY` clause groups the input tuples into partitions. Each partition is treated independently and partitions do not influence each other. The `ORDER BY` clause defines an order within each partition and thereby introduces a notion of neighboring tuples. The remaining parts of the `OVER` clause allow to, for each tuple, select a subset of those neighboring tuples, the *window frame*.

Window frames can express a wide variety of queries:

```
rows between unbounded preceding and current row
```

computes a running aggregate

```
range between '1 month' preceding and current row
```

computes a sliding aggregate, and

```
rows between unbounded preceding and unbounded following
exclude current row
```

can compare a row against, e.g., the maximum of all other rows. Depending on the framing mode, the frame boundaries are identified as a fixed row offset (`ROWS`) or based on a value range (`RANGE`). Additionally, the frame exclusion options (`EXCLUDE CURRENT ROW`, `EXCLUDE TIES`, ...) allow excluding tuples explicitly.

While frame boundaries are frequently provided as constants, they can be arbitrary expressions. One example use case for this flexibility is stock market limit orders that are only valid for a time interval chosen by the individual traders. To figure out which orders executed at a favorable time, one can compare them with all other orders during the `good_for` interval using

```
select price > median(price) over (
  order by placement_time
  range between current row and good_for following)
from stock_orders
```

While non-constant boundaries are not very common, SQL provides for this flexibility and a full-fledged system should support them.

2.3 Window Functions

Based on partitioning, sort order and framing, the window function computes the result value for each tuple. The available window functions can be divided into three classes:

- Most *aggregate functions* (`AVG`, `MIN`, ...) can also be used as window functions.
- *Value functions* (`NTH_VALUE`, `FIRST_VALUE`, ...) evaluate a given expression against a neighboring tuple in the partition.
- *Rank functions* (`RANK`, `ROW_NUMBER`, `CUME_DIST`, ...) rank a row in comparison to the other tuples in the partition.

The usage of sort order and framing depend on the window function. For aggregate functions, the aggregate is applied to all input tuples within the current frame. The ordering is only used to establish the window frame. Value functions use the ordering for two purposes: As for aggregate functions, it is used to establish the window frame. Additionally, the ordering is reused to define the semantics of the value function itself, i.e. the criterion by which `NTH_VALUE` selects a tuple. Rank functions do not support framing and only use the `ORDER BY` clause to define the ranking order.

2.4 Proposed Extensions

As just described, the SQL standard distinguishes between window frames and window functions which can be combined to answer a large variety of analytical questions. At the same time, the framing functionality is limited to a small set of functions. Holistic aggregates like percentiles and window functions like rank cannot be used with a window frame. We propose to lift this restriction.

For DISTINCT aggregates this is straightforward. We can simply write COUNT(DISTINCT x) OVER(...). In contrast, all order-based aggregate and window functions need a second ORDER BY clause, such that the window function can use a different ordering than the frame. The placement of the additional ORDER BY clause follows the convention established by the SQL standard for other order-sensitive aggregates like ARRAY_AGG. The ORDER BY in the OVER clause is then used exclusively for establishing the window frame, while the order attached to the window function is used to define the function's result. For many systems, those extensions do not require grammar changes: E.g., the PostgreSQL grammar [4] accepts DISTINCT and ORDER BY as part of every function call, and only rejects those clauses during semantic analysis in places where those keywords do not apply.

With those extensions, the framing options from the OVER clause are now fully composable with all window functions, as demonstrated in the following example: When analyzing TPC-C performance, comparing performance numbers achieved years ago against today's performance numbers does not represent how much of an achievement those numbers were back in the days. A fair judgment requires comparing a TPC-C result with all *previous* performance numbers while excluding later submissions. The query

```
select dbsystem, tps,
  count(distinct dbsystem) over w,
  rank(order by tps desc) over w,
  first_value(tps order by tps desc) over w,
  first_value(dbsystem order by tps desc) over w,
  lead(tps order by tps desc) over w,
  lead(dbsystem order by tps desc) over w,
from tpcc_results
window w as (order by submission_date
  range between unbounded preceding and current row)
```

shows how to use the proposed extensions for such an analysis. For each entry from the tpcc_results table, the window frame w only includes previously submitted performance results. Based on that window frame, the query computes the number of competing systems, the rank which the performance result achieved compared to previous systems, the performance and name of the overall best system (FIRST_VALUE) at the time of submission, and the next-best system's performance (LEAD), so that one can judge how tight the ranking was. Similar queries come up in almost all contexts when putting historical performance data into context: TeraFLOPS of graphic cards, marathon times of athletes, fuel efficiency of automobiles. Without our proposed extensions, those queries require a complicated and – as shown empirically in Section 6.2 – inefficient SQL formulation.

The newly gained functionality poses two challenges, though: For distinct aggregates, we need to efficiently deduplicate tuples. For the other window functions, we have two independent ORDER BY clauses, but can physically sort the tuples only by one of them.

3 RELATED WORK

This paper is related to three lines of work: window processing in OLAP systems, window processing in streaming systems and range queries in the algorithm community. This section first summarizes the contributions of each line of research. Next, the strategies employed by previous work are discussed in order to identify their main choke points and guide the design of merge sort trees.

3.1 Lines of Research

In the context of OLAP systems, Leis et al. [27] showed how to efficiently implement window functions and introduced segment trees as an index structure for distributive aggregates. Kohn et al. [24] and Cao et al. [11] both improved on this by avoiding duplicated work across multiple window functions (e.g., reusing partitioning and ordering between window functions). The work presented in our paper is orthogonal and can be combined with this previous work. Wesley and Xu pushed the boundaries of the SQL standard and introduced incremental algorithms for framed distinct counts, percentiles and modes [38]. Their algorithms for percentiles and modes were recently incorporated into DuckDB.

For windowing in stream-processing systems, a multitude of data structures (e.g., base intervals [1], SlickDeque [30], FlatFAT [34], FIBA [33]) and aggregation strategies (e.g., Cutty [12], Scotty [36]) were proposed, covering a large variety of frame types, in-order and out-of-order arrivals, as well as window sharing across multiple queries. But very few papers consider holistic aggregates: The only technique we are aware of is base intervals [1] which can compute percentiles in $O(n(\log n)^2)$ by storing sorted lists of input values in the aggregation states.

In the algorithms community, window functions are referred to as *range queries*. *Range medians* were extensively studied by multiple authors [9, 10, 19, 23, 25]. The proposed algorithms vary depending on how many medians are queried, whether the requested medians are known upfront, and whether an index can be built upfront. Applied to the use case of SQL window functions, all those algorithms take $O(n \log n)$ time and $O(n \log n)$ space[1]. In comparison to this previous work, our merge sort tree is a static, more compact data structure and can be represented more efficiently without pointers. An $O(n \log n)$ algorithm for *range distinct counts* was previously presented [31]. Holistic ranks and value functions can be computed in $O(n \log n)$ using a suitably annotated tree [17], also called *order statistic tree*. However, an order statistic tree cannot be efficiently parallelized. For windowed modes, algorithms exist which take $O(n \log n)$ time and $O(n)$ space [13, 25].

Our paper combines merge sort trees and range trees by Bentley [6, 7] with a variation of the fractional cascading technique by Chazelle et al. [14, 15] and extends them with preprocessing steps to make them applicable to holistic aggregates in SQL.

[1]Some publications [9, 10, 23] claim $O(n)$ space consumption *in the word RAM model*. Note that the word-RAM model assumes that the word size grows with the problem size, and is as such not relevant for SQL types with fixed integer widths.

Table 1: State-of-the-art algorithms for holistic aggregates compared to our proposed merge sort tree algorithm.

aggregate	algorithm	serial runtime	space	parallel
dist. count	incremental [38]	$O(n)$	$O(n)$	no
	MST (our approach)	$O(n \log n)$	$O(n \log n)$	yes
dist. aggr.	naive	$O(n^2)$	$O(n)$	no
	MST (our approach)	$O(n \log n)$	$O(n \log n)$	yes
percentile	incremental [38]	$O(n^2)$	$O(n)$	no
	segment tree [1, 27]	$O(n(\log n)^2)$	$O(n \log n)$	yes
	order statistic tree [17]	$O(n \log n)$	$O(n \log n)$	no
	MST (our approach)	$O(n \log n)$	$O(n \log n)$	yes
rank	order statistic tree [17]	$O(n \log n)$	$O(n \log n)$	no
	MST (our approach)	$O(n \log n)$	$O(n \log n)$	yes

3.2 Existing Evaluation Strategies

Table 1 summarizes the state-of-the-art algorithms for holistic aggregates and compares them against our proposed merge sort tree for SQL's default window frame `BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. The table contains the complexity for computing all n output tuples. Depending on the specific aggregate, the current state-of-the-art algorithms come with different drawbacks. Merge sort trees (MSTs) win over most competing algorithms because they can be parallelized. Compared to segment trees, i.e., the only parallelizable competitor, merge sort trees provide better runtime complexity. For distinct counts and percentiles, a decision in favor of merge sort trees needs further consideration, though: The incremental algorithm from Wesley and Xu [38] computes percentiles in $O(n)$ space, while merge sort trees use $O(n \log n)$ space. We deem the higher memory consumption a worthy tradeoff because a $O(n^2)$ runtime limits the solvable problem size more severely than a $O(n \log n)$ memory usage. For distinct counts, the tradeoff between a serial $O(n)$ algorithm and a parallel $O(n \log n)$ algorithm depends on the input size and the number of available of cores. In practice, the window function is part of a window operator which first needs to run a sorting step to establish the window frame order. If this sorting step is comparison-based and thereby $O(n \log n)$, sorting dominates the overall runtime, and the better asymptotic complexity of Wesley and Xu's algorithm does not yield a better end-to-end complexity.

The reason why most competitors cannot be parallelized is that they keep an aggregation state up-to-date as tuples enter or leave the window frame. When a second thread wants to process the second half of input tuples, it first needs to compute the aggregate state at the starting point. It thereby re-does all the work of the first thread before processing its own input tuples. Even worse, modern parallel systems use task-based parallelism instead of thread-based parallelism [26]. Work is divided into a much larger number of independent tasks than available threads. Usually, each task has a fixed size and the number of tasks linearly depends on the number of input tuples. In such a task-based model, parallelization actively deteriorates the performance of our competitors to $O(n^2)$ or worse (depending on the aggregate) because each of the $O(n)$ tasks needs to first aggregate all $O(n)$ preceding tuples. In general, the amount of duplicate work between tasks and thereby the impact of parallelization on our competitors depend on the requested frame size. The larger the frame size, the worse task-based parallelization impacts the runtime of our competitors. Also, for non-monotonic frame boundaries, as in the example on stock limit orders, all competitors except the segment tree are $O(n^2)$ even in the serial case. This is because the same input tuple might enter and leave the window frame multiple times. In the worst case, each input tuple enters the window frame $O(n)$ times, pushing the computation cost per result tuple from $O(1)$ to $O(n)$.

Previous work used segment trees to parallelize the computation of distributive and algebraic aggregates [27]. Segment trees depart from the idea of keeping an aggregate-state up-to-date. Instead, segment trees are first constructed in a parallelized $O(n)$ build phase before being used as read-only index structures during an embarrassingly parallel $O(n \log n)$ probe phase. As an added benefit, the runtime stays unchanged even for non-monotonic window frames because segment trees do not rely on overlap between consecutive window frames. This $O(n \log n)$ time complexity depends on the fact that two aggregation states for distributive and algebraic aggregates can be merged in $O(1)$. Previous work [1] applied segment trees to percentiles by annotating each node of a segment tree with a sorted list. The resulting algorithm is indeed parallelizable, but is $O(n(\log n)^2)$ because two aggregation states cannot be combined in $O(1)$ and combining all relevant states from the segment tree into an overall aggregate result takes $O((\log n)^2)$. Similarly, the aggregation state for all other holistic aggregates cannot be merged in $O(1)$. Due to this non-constant merge costs, segment trees and other approaches based on combining partial aggregates (such as prefix sums [8], FIBA [33], FlatFAT [34], Cutty [12], Scotty [36]) cannot be efficiently applied to holistic aggregates.

Besides their better algorithmic properties, merge sort trees have practical implementation benefits: With merge sort trees, a single data structure covers a wide range of holistic aggregates. This reduces implementation effort compared to multiple specialized algorithms for the various aggregates. Furthermore, the most demanding part of parallelizing merge sort trees is the parallel merging of sorted lists. Such a parallelized merge is usually already present in database systems as part of a merge join or as part of the `ORDER BY` implementation and can be reused. Note that even if a database system employs quicksort or some other sorting algorithm, proper parallelization of a quicksort still requires a merge step to combine the thread-local sorted lists into a single sorted result [16, 18].

4 EVALUATING HOLISTIC AGGREGATE WITH MERGE SORT TREES

In this section, we show how to use merge sort trees to efficiently compute framed holistic aggregates under parallelization. We first provide an overview of our general approach and motivate it based on the shortcomings of previously proposed algorithms. While the merge sort tree data structure is identical for all holistic aggregates, the different aggregates require different preprocessing steps and query the tree in different ways. We use `COUNT DISTINCT` to motivate the necessary operations on a merge sort tree before introducing the data structure itself. We then describe the preprocessing and query methods for other aggregates.

4.1 High-Level Overview

To arrive at an efficient, parallelizable algorithm, we adopt the same strategy previously applied by Leis et al. for segment trees [27]: We

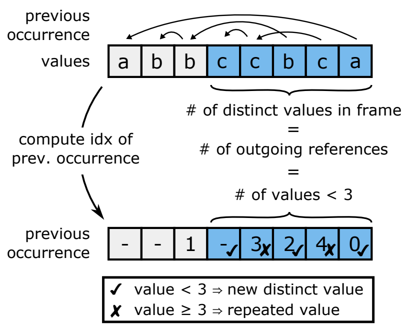
previous occurrence
values
a b b c c b c a
# of distinct values in frame = # of outgoing references = # of values < 3
compute idx of prev. occurrence
previous occurrence
- - 1 -✓ 3✗ 2✓ 4✗ 0✓
✓ value < 3 ⇒ new distinct value
✗ value ≥ 3 ⇒ repeated value

Figure 1: Computing a windowed COUNT DISTINCT by evaluating a 2-dimensional range query.

split the evaluation into two phases. In the first phase, we build an index structure. In the second phase, we use this index to efficiently compute all aggregate results. To achieve a runtime of $O(n \log n)$, we replace segment trees with *merge sort trees*. Segment trees rely on combining partial aggregates to form the final aggregate results. For holistic aggregates, combining partial aggregates is expensive, though. Merge sort trees sidestep those costs by directly computing the aggregate result without combining partial aggregates.

Building the merge sort tree takes $O(n \log n)$ time and the build phase can be parallelized using existing algorithms for parallel sorting. After the build phase, the tree can be queried for the aggregate result for any window frame in $O(\log n)$ time. As there is one query for each of the n input tuples, the overall window operator can be evaluated in $O(n \log n)$. During the query phase, the merge sort tree is not modified and shared between threads, making the query phase embarrassingly parallel. Just as segment trees, merge sort trees do not rely on overlap between consecutive window frames, and guarantee $O(n \log n)$ runtime even for non-monotonic frames.

4.2 Windowed COUNT DISTINCT

`COUNT DISTINCT` aggregates are usually computed using hash tables: A hash table is used to deduplicate all input values, and the number of entries in the resulting hash table equals the `COUNT DISTINCT`. Recomputing this hash table from scratch for every window frame is prohibitively expensive, though: Even for simple frame boundaries like `UNBOUNDED PRECEDING AND CURRENT ROW`, this algorithm has complexity $O(n^2)$. Wesley and Xu [38] hence proposed an incremental $O(n)$ algorithm which updates the hash table for consecutive window frames. But, as discussed in the previous section, this approach comes at the expense of parallelization. In fact, under task-based parallelism the algorithm is back at $O(n^2)$.

Using range counting queries to compute distinct counts: To enable parallelization, we need to approach the problem from a different angle. Instead of using a hash table to identify duplicates, we preprocess the input as shown in Figure 1. The top half of the picture shows 8 input tuples, with 3 distinct values a, b and c. In this example, the queried window frame (highlighted in blue) consists of the last 5 values, and as those 5 values contain all 3 distinct values, the `COUNT DISTINCT` yields "3" on this window frame.

Above the input tuples, the figure shows backreferences pointing from each value to its previous occurrence. The key insight is: The number of distinct values inside a window frame always equals the number of backreferences originating inside the window frame and pointing before the frame's start. This is explained by the following reasoning: Every value which is observed for the first time inside the frame has a backreference pointing before the start of the frame. Every duplicate value already appeared within the window before and its backreference hence points to a position inside the window frame. By only counting references which point before the frame's start, we are only counting the first occurrence of each value and avoid counting duplicated values multiple times.

The bottom half of Figure 1 shows an alternative representation of those backreferences: Backreferences are represented as an array of integers where each entry contains the index of the previous occurrence of the corresponding value. The first two entries of the array have the special value "-" because the corresponding values (a and b) appear for the first time, the third entry is 1 because the corresponding value b previously occurred at index 1, and so forth.

Algorithm 1 Computing the index of the previous occurrence.

```
1:  function ComputePrevIdcs(in)
2:      sorted ← []
3:      for i ← 0 to in.size do
4:          sorted[i] ← (in[i], i)
5:      Sort sorted lexicographically increasing        ▷ Stable Sort
6:      prevIdcs ← []
7:      prevIdcs[0] ← "-"
8:      for i ← 1 to in.size do
9:          if sorted[i].first == sorted[i-1].first then
10:             prevIdcs[i] ← sorted[i-1].second
11:         else
12:             prevIdcs[i] ← "-"
13:     return prevIdcs
```

This index array can be computed in parallel using Algorithm 1. The algorithm constructs a copy of the input data and annotates each element with its original position. Next, the copy is sorted lexicographically, i.e., by the input value, using the original position as a tie breaker for identical values. The sort step is effectively a stable sort on the input value, leaving the relative order of duplicates unchanged. All duplicated input values are thereby grouped together into continuous runs, with each run being sorted ascendingly by the original positions. After sorting, the previous occurrence, if any, of the value `sorted[i]` is at `sorted[i-1]`. We can now compute the final `prevIdcs` array in a linear pass over the sorted data. Both loops are $O(n)$ and trivial to parallelize. Sorting is $O(n \log n)$ and there is plenty of research on parallel sorting.

The complete construction of the index array does not involve the actually queried window frame. The window frame only becomes relevant to compute the final distinct count: To compute the distinct count for a window from index a until index b, we count the entries smaller than a within `prevIdcs[a...b]`. The window frame in Figure 1 starts at index 3, hence we are looking for entries smaller than 3 within the highlighted input section. All qualifying entries are annotated with a small check mark in the figure.

A naive linear pass over the array to count all such entries would take $O(n)$ per distinct count, and thereby push the overall processing time for all n output tuples to $O(n^2)$. We need a more efficient way to count those entries.

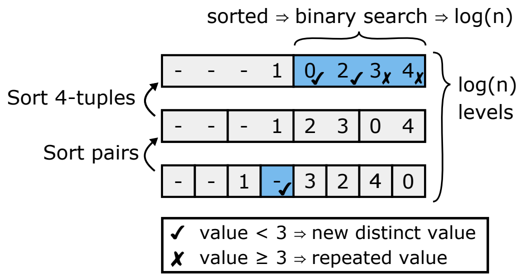
sorted ⇒ binary search ⇒ log(n)
Sort 4-tuples
Sort pairs
log(n) levels
✓ value < 3 ⇒ new distinct value
✗ value ≥ 3 ⇒ repeated value

Figure 2: A merge sort tree improves query time to $O(n(\log n)^2)$ by utilizing a tree of sorted lists.

Another way to think of this counting is as a two-dimensional range query: The first dimension is sorted by the order of the window frame and filtering on this dimension filters the tuples down to the current window frame. The second dimension is sorted by the index of the previous occurrence and filtering on this dimension excludes duplicate values. This maps the computation of a distinct count to a two-dimensional range counting query. We now only need an efficient data structure for such range counting queries.

Merge sort trees can efficiently answer such two-dimensional range counting queries. To arrive at a better asymptotic complexity, we need a way to count the number of indices smaller than the given threshold without comparing each individual entry against the threshold. An obvious solution would be to sort the array and then use a binary search to locate the threshold value inside the sorted array. Sorting the array by `prevIdcs` is not possible, though. To compute a framed distinct count, we have to count how many entries *inside the window frame* are smaller than the threshold. The array is already sorted by the `ORDER BY` of the window frame, and we rely on this order to place all values inside the window frame into the consecutive range `prevIdcs[a...b]`. Sorting the `prevIdcs` array by another criterion would destroy this sort order and inhibit us from efficiently identifying all tuples inside the window frame. Instead, we need to keep both sort orders at the same time.

To solve this issue, we store the same list multiple times, as visualized in Figure 2. The lowest layer stores the original `prevIdcs` array. On top of that, there are multiple partially sorted copies. In the first copy, pairs are sorted; the second copy contains sorted runs of length 4. For larger input arrays, the tree would have additional layers, until all numbers are sorted inside a single run. In general, the tree has $O(\log n)$ layers and needs $O(n \log n)$ memory. Such a tree can be efficiently built bottom-up in $O(n \log n)$ by modifying a merge sort algorithm to preserve the intermediate sorted runs instead of discarding them. Because of this resemblance of the data structure with the intermediate results of a merge sort, we call the resulting data structure a *merge sort tree.*

One can now use this merge sort tree to efficiently count the number of values below a certain threshold in a given window frame. To do so, the queried range is pieced together from the various sorted lists. Figure 2 shows such a query for the range $[3; 7]$. The queried range can be pieced together using the sorted runs $[3; 3]$ on the lowest level and $[4; 7]$ on the first level. For each range, we can determine how many values are smaller than the given threshold using a binary search on the corresponding sorted run. In this example, we execute 2 binary searches.

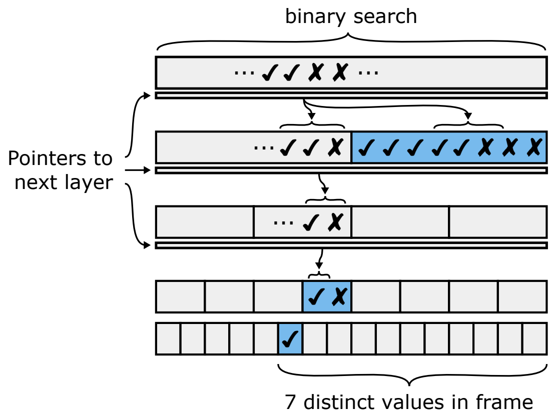

Figure 3: Fractional cascading: Pointers between layers restrict the search ranges on lower layers, reducing query time to $O(n \log n)$.

In general, we need at most 2 binary searches per layer: On the highest level, we have always exactly one sorted run. On all other levels, we need at most two runs, one run to the left and one run to the right of the range already covered on the higher levels. As there are $O(\log n)$ layers and each binary search has $O(\log n)$ execution time, the evaluation of a distinct count for a given window frame is $O((\log n)^2)$. For the overall evaluation of a window query, this gives us an $O(n(\log n)^2)$ execution time. The construction costs of $O(n \log n)$ for the merge sort tree are dominated by this query time. We hence need to focus our attention on querying the merge sort trees to further improve the overall execution time.

Fractional cascading improves this query time to $O(n \log n)$ by avoiding to re-run the binary search on each tree level. The key idea is to traverse the tree top-down and reuse the result of the binary search on one level to narrow down the search range on the next level [14, 15]. Figure 3 illustrates this approach. During construction of the merge sort tree, we annotate the tree with additional pointers. When querying the tree to determine a distinct count, we perform a $O(\log n)$ binary search only on the highest level of the merge sort tree. The additional pointers then allow us to reuse the result of this binary search to narrow down the search range on the next finer-grained level. On the next level, the binary search profits from the reduced search range to the point that it runs in constant time. When descending to the next level, we again reuse the binary search from the now current level, and thereby turn all except for the first binary search into $O(1)$ operations. In sum, we need $O(\log n)$ for the first binary search plus $O(1)$ on each of the $O(\log n)$ levels, leading to an overall complexity of $O(\log n)$ per query. The initial top-level binary search *cascades* down from the top to the bottom.

To ensure $O(1)$ time for each of the searches on the lower levels, one can annotate each element on each level with two additional pointers as visualized in Figure 4. For each element on the upper level, the pointers locate an element's sort position on the next level in the left and right sublist. The pointers point to the largest element in the sublist which is smaller or equal to the annotated element. Having located an element on the upper level, one can now find the element's positions on the next level by simply traversing the pointer. The additional pointers increase the memory consumption

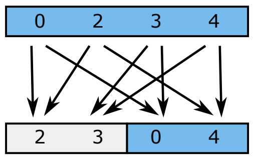

Figure 4: Pointers between two tree levels.

only by a constant factor; the asymptotic memory consumption stays unchanged at $O(n \log n)$. The pointers are computed as a byproduct of constructing the merge sort tree by persisting the input iterators used during the merge steps.

Instead of annotating every element, one can also annotate only every kth element. As long as k is chosen as a constant, this sampling does not influence the asymptotic complexity. Between one annotated element and the next annotated element, there are always at most k elements, and on the next level we have to locate our element again by searching within those k elements. This still yields a constant upper bound on the size of the search range, thereby ensuring that each tree level can be traversed in $O(1)$. While asymptotic complexity is not influenced by reducing the number of annotated elements, this optimization is relevant for a practical implementation as discussed in Section 5.1.

This approach was inspired by and is closely related to a technique known as *fractional cascading*: Fractional cascading was originally introduced by Chazelle et al. as a technique to solve computational geometry problems, such as the intersection of paths and lines and various types of range searches [14, 15]. It allows searching a set of *unrelated* lists by injecting additional sentinel values into the lists and then annotating *every* list element with pointers into the related lists. Our technique improves on this in two key aspects: First, in our case the lists inside the upper layers were created by merging lists from the lower layers. Hence, we do not need to inject additional sentinel values, but only need to annotate existing values. Second, we do not need to annotate *every* list element with pointers, but instead relax it to only annotate every kth element.

4.3 Arbitrary distinct aggregates

Next, we expand the algorithm to handle arbitrary distinct aggregates. Without loss of generality, the rest of this section focuses on a `SUM DISTINCT` aggregate. The presented approach is applicable to all distributive aggregates.

To compute a windowed `SUM DISTINCT`, we annotate a merge sort tree as shown in Figure 5: Each element is annotated with the distinct sum of the values for all entries up to and including the current position. Using those partial sums, the overall `SUM DISTINCT` for a frame can be computed by (1) covering the frame with sorted runs of the merge sort tree, (2) searching the lower bound of the window frame inside each sorted run, and (3) adding up the corresponding partial sums

The first two steps are identical to computing a windowed distinct count. The novelty for arbitrary aggregates consists of the third step: Instead of only counting the number of values smaller than the given lower bound, we combine the partial aggregates associated with this lower bound.

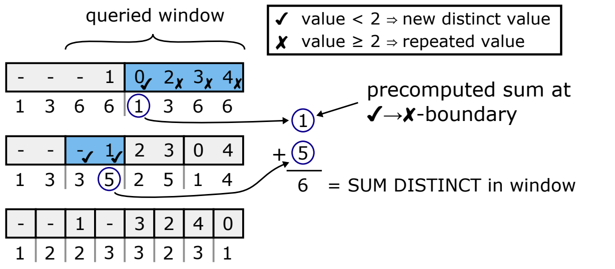
queried window
✓ value < 2 ⇒ new distinct value
✗ value ≥ 2 ⇒ repeated value
- - - 1 0✓ 2✗ 3✗ 4✗
1 3 6 6 1 3 6 6
precomputed sum at ✓→✗-boundary
1
- - -✓ 1✓ 2 3 0 4
1 3 3 5 2 5 1 4
+ 5
6 = SUM DISTINCT in window
- - 1 - 3 2 4 0
1 2 2 3 3 2 3 1

Figure 5: Computing a windowed SUM DISTINCT from an annotated merge sort tree.

As before, finding the positions inside all relevant runs takes $O(\log n)$ thanks to fractional cascading. To compute the aggregate result, at most $O(\log n)$ partial aggregates need to be combined. This leads to an overall complexity of $O(\log n)$ for computing a single windowed `SUM DISTINCT` value and thereby leaves the complexity of the complete window operator unchanged at $O(n \log n)$.

Note that the proposed algorithm works for all distributive or algebraic aggregates, including user-defined aggregates as long as they provide a function to merge two aggregation states. An inverse function to the merge function, i.e. a function to remove a value from an aggregate state, is not required. This is an important benefit of the proposed algorithm because for many user-defined aggregates such an inverse function is not available.

4.4 Windowed Rank Functions

Just like distinct aggregates, windowed rank functions can be reduced to two-dimensional range queries. For that purpose, let's recall the syntax for windowed rank functions from Section 2.4:

```
select dbsystem, tps,
  rank(order by tps desc) over w
from tpcc_results
window w as (order by submission_date
  range between unbounded preceding
    and current row)
```

This query contains two `ORDER BY` clauses: The `ORDER BY` within the function call determines the criterion by which competitors are ranked. The `ORDER BY` within the window definition is used to establish the window frame and thereby determines which competitors to rank the current row against. The rank of a row equals the number of tuples within the window frame which compare smaller than the current row for the rank's `ORDER BY` criterion.

As such, the algorithmic core challenge is the same as for a distinct count: We need to count the number of tuples from our window frame which are smaller than a given threshold. Therefore, we reuse the merge sort tree introduced in the previous section. If we build the merge sort tree directly with the unmodified tuple values as the lowest layer, a merge sort tree allows us to efficiently determine the number of values smaller than a given value inside an arbitrary window frame in $O(\log n)$ using the same algorithm as in Section 4.2. As for distinct counts, we can thereby evaluate the complete window operator in $O(n \log n)$.

The `ROW_NUMBER` function can be computed by disambiguating duplicate elements based on their position in the input data, such that two elements never compare as equal. Based on this, the rank functions PERCENT_RANK and NTILE are straightforward to implement: PERCENT_RANK is a scaled version of RANK, and CUME_DIST is a scaled version of ROW_NUMBER.

DENSE_RANK is the only rank-related function from the SQL standard which cannot be answered by a merge sort tree. A dense rank requires counting the number of *distinct* tuples within the window frame which compare smaller than the current row. As for distinct counts, the distinctness criterion can be turned into a range query by using the indices of a value's previous occurrence. In combination with the two existing ordering criteria, this leads to a 3-dimensional range query. A merge sort tree can only handle 2-dimensional range queries, though. Instead, a structure which supports higher-dimensional range queries is needed. *Range trees* offer such functionality. We do not discuss range trees in this paper and refer the reader to [6, 7] for more details. A d-dimensional range tree takes $O(n(\log n)^{d-1})$ space and, if combined with fractional cascading, an individual range count query takes $O(n(\log n)^{d-1})$ time. With a range tree, a framed DENSE_RANK window query can be evaluated in $O(n(\log n)^2)$ time using $O(n(\log n)^2)$ space.

4.5 Percentiles and Value Functions

Looking back at our motivating examples for framed percentiles

```
select l_shipdate,
  percentile_disc(
    0.99, order by l_receiptdate - l_shipdate
  ) over w
from lineitem
window w as (order by l_shipdate
  range between '1 week' preceding and current row)
```

and value functions

```
select dbsystem, tps,
  first_value(tps order by tps desc) over w,
  first_value(dbsystem order by tps desc) over w
from tpcc_results
window w as (order by submission_date
  range between unbounded preceding and current row)
```

we see that they again specify two ORDER BY clauses. The ORDER BY in the function call defines the criterion by which a value is selected. The ORDER BY in the OVER clause establishes the frame.

Both percentiles and value functions map to the same underlying problem: Selecting the ith smallest value. For value functions, i is directly provided in the query. For the pth percentile, i is computed as $s * p/100$ where s is the number of elements inside the frame.

To do so, we build on the following idea [9, 19]: We first preprocess the original input to create a permutation array as depicted in Figure 6. The input tuples are originally sorted by the window frame's sort criterion, as shown in the top half of the figure. We then sort the data by the criterion for the percentile, in this example by alphabetic order. During the sorting, we keep track of the original positions and then store the original indices in the permutation array shown at the bottom half of the figure.

Based on that permutation array, one can find the ith smallest value within any window frame by finding the ith index pointing into the frame. Figure 6 visualizes this for a median query on the range $[2; 6]$. There are 5 values in that range, so the median is the

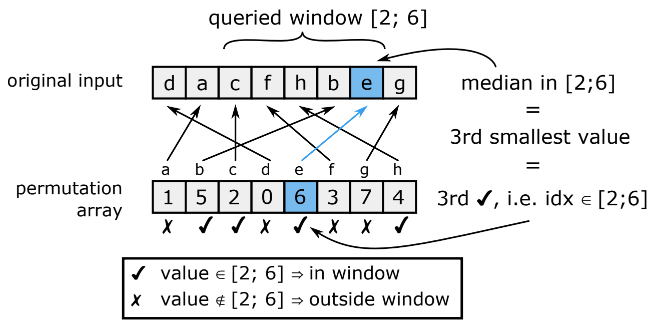

Figure 6: Finding a windowed median by scanning the permutation array.

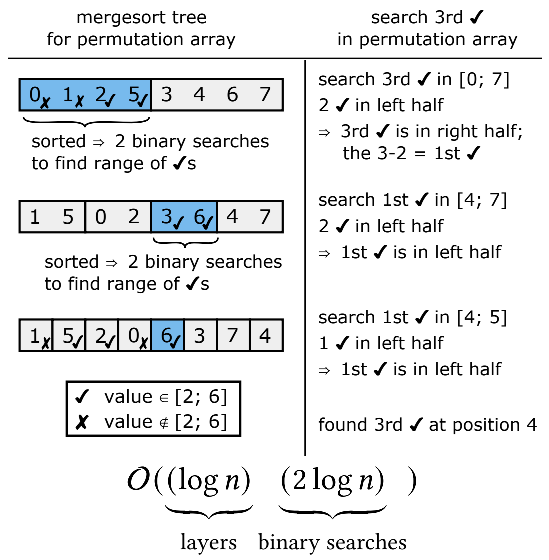
mergesort tree for permutation array
search 3rd ✓ in permutation array
0✗ 1✗ 2✓ 5✓ 3 4 6 7
sorted ⇒ 2 binary searches to find range of ✓s
search 3rd ✓ in [0; 7]
2 ✓ in left half
⇒ 3rd ✓ is in right half; the 3-2 = 1st ✓
1 5 0 2 3✓ 6✓ 4 7
sorted ⇒ 2 binary searches to find range of ✓s
search 1st ✓ in [4; 7]
2 ✓ in left half
⇒ 1st ✓ is in left half
1✗ 5✓ 2✓ 0✗ 6✓ 3 7 4
search 1st ✓ in [4; 5]
1 ✓ in left half
⇒ 1st ✓ is in left half
✓ value ∈ [2; 6]
✗ value ∉ [2; 6]
found 3rd ✓ at position 4
$O(\underbrace{(\log n)}_{\text{layers}} \underbrace{(2\log n)}_{\text{binary searches}})$

Figure 7: A merge sort tree speeds up the search in the permutation array from $O(n)$ to $O((\log n)^2)$. Fractional Cascading reduces this further to $O(\log n)$.

3rd smallest value. We iterate over the permutation array from left to right. The first value "a" occurred outside the window frame at index 1 and therefore we skip it. The next two values "b" and "c" fall within the frame. Those are the two smallest values from the frame, but since we are looking for the 3rd smallest value we keep going. The value "d" is outside the frame and skipped. The next value "e" is in the frame, and given we already encountered two qualifying values, this is the 3rd smallest value and thereby the median.

Finding this index with a linear search would take $O(n)$ and since we have to repeat it for each tuple of our window operator the overall query time would be $O(n^2)$. To improve upon that time, we can employ a merge sort tree. The merge sort tree is constructed as usual, but with the indices from the permutation array as the lowest layer. Searching for the ith value inside this tree can be achieved in $O(\log n)$ as sketched in Figure 7: To find the searched element one starts at the top of the tree. The searched element might be either in the left or right sublist. To decide on which side the searched element is, we count the number of qualifying elements in the left sublist. One can do so by using two binary searches to locate the upper and the lower frame bound inside the sorted list. In case the number of qualifying elements in the left sublist is smaller than the searched element, we descend to the left, otherwise to the right. Those steps are repeated until reaching the lowest layer. On the lowest layer, the remaining range consists of exactly one element: the searched element. The tree has $O(\log n)$ layers and on each layer we do 2 binary searches. This would leave us with $O((\log n)^2)$ complexity. But, as before, fractional cascading can avoid the repeated binary searches and thereby provides an execution time of $O(\log n)$.

As an additional semantic requirement, percentiles ignore `NULL` values while value functions by default do not consider `NULL`s when selecting the ith value. For value functions the SQL standard provides the `IGNORE NULLS` clause to exclude `NULL`s. As such, we need a way to exclude `NULL` tuples. Doing so is possible by building the merge sort tree only on the non-ignored tuples and remapping indices between the original input of the window operator and its filtered representation. This additional step is $O(n)$ and hence does not influence the overall asymptotic complexity.

In summary, our algorithm for percentiles and value functions consists of 3 steps:

(1) Compute the permutation array $\rightarrow$ $O(n \log n)$
(2) Build the merge sort tree $\rightarrow$ $O(n \log n)$
(3) Compute the percentile for all n input tuples $\rightarrow$ $O(n \log n)$

Summing up all steps, we arrive at an overall execution time of $O(n \log n)$.

4.6 LEAD and LAG

The functions `LEAD` and `LAG` can be extended to a windowed version with two independent `ORDER BY` clauses just like `first_value` and other value functions.

To evaluate a windowed `LEAD`/`LAG`, one has to (1) compute the `ROW_NUMBER` of the own row, (2) adjust the row number by adding or subtracting an offset, (3) find the row at that offset, and (4) evaluate the expression provided to `LEAD`/`LAG` on this row. One can use the algorithm from Section 4.4 to determine the row number of the own row (step 1) and the algorithm from Section 4.5 to find the row with the adjusted position (step 3). Both algorithms are in $O(n \log n)$, so the overall algorithm for `LEAD`/`LAG` is also $O(n \log n)$.

4.7 Non-continuous Window Frames

So far, all presented algorithms relied on the fact that a window frame consists of a single continuous range of rows. But not all window frames are continuous. A `FILTER` clause or a frame exclusion clause can lead to non-continuous frames. The merge sort tree is also applicable for such non-continuous frames.

In the presence of a frame exclusion clause (`EXCLUDE CURRENT ROW`, `EXCLUDE TIES`, ...), a window frame can contain up to two holes, breaking the otherwise continuous range into three continuous ranges. The construction of the merge sort trees stays unchanged, but the check for frame inclusion when querying the tree becomes more complex. This does not influence the asymptotic complexity because all frames expressible with frame exclusion clauses can be pieced together from at most three ranges, i.e., a constant number.

The `FILTER` clause from SQL:2003 allows excluding individual rows during aggregation by writing `SUM(a) FILTER(a > 0) OVER(...)`. This clause can also be extended to distinct aggregates

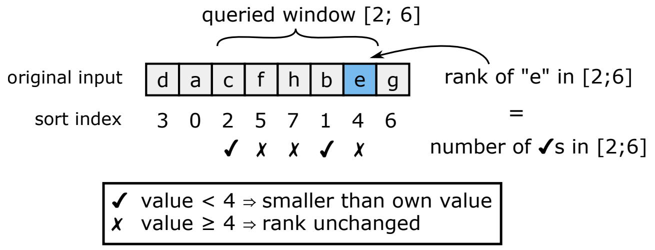

Figure 8: Preprocessing applied for rank queries.

and all other window functions discussed in this paper, by supporting the syntax `RANK(ORDER BY a) FILTER(is_active) OVER(...)`. The key insight here is: Whether a particular row is excluded or not depends only on the tuple itself and can be determined upfront, independently of the window frame. As such, the `FILTER` clause can be supported with the same trick we used for `IGNORE NULLS` in Section 4.5: We skip inserting filtered-out tuples into the merge sort tree in the first place and remap indices accordingly.

5 IMPLEMENTATION DETAILS

We integrated our algorithms into Hyper, a commercial high performance database system which employs code generation. This section gives an overview of our implementation choices and presents additional optimizations.

5.1 In-Memory Representation

We implemented merge sort trees as pure in-memory structures. If necessary, they could also be spooled to disk. We represent the tree using contiguous integer arrays, one array per tree level. Pointers inside the merge sort tree are expressed as indices into the array of the next tree level. Payload values are represented as integers through additional preprocessing.

The payload values of the tree for percentile queries are already integers. For distinct aggregates, *prevIdcs* contains mostly integer indices, but also the special value "-". We map the special value "-" to 0 and shift all other indices by 1. For rank functions, we number the input tuples with dense integer numbers representing their sort order as visualized in Figure 8. This preprocessing can be done using a sort algorithm in $O(n \log n)$. Thereby, we avoid handling all SQL types and intricacies of `ORDER BY` clauses (multiple sort criteria, sorting by arbitrary SQL expressions, collation-aware string comparisons, `NULLS LAST`, ...) as part of the merge sort tree and instead move this complexity into the preprocessing step.

Depending on the input size, we build the tree using either 32-bit or 64-bit integers. This decision is made independently for each window partition at runtime. Using smaller integer widths reduces memory consumption and, thanks to reduced memory bandwidth usage, also improves performance.

For the same reasons, we annotate only every kth element with pointers into the next level. As discussed in Section 4.2, this sampling does not deteriorate asymptotic complexity. Storing fewer pointers reduces strain on memory bandwidth. On the other hand, a larger k increases the number of comparisons necessary on each tree level. By choosing k, one can thereby tradeoff memory consumption and memory bandwidth against CPU cycles. In addition to reducing the number of pointer annotations, we use a higher fanout f for our tree. During the build phase of the tree we hence do multi-way merges. For larger fanouts, a query has to do more binary searches per layer in the tree. Those additional searches do not deteriorate asymptotic complexity because the number of binary searches per layer is still bounded by $2f$. The overhead for additional binary searches is offset by the lower memory consumption and better cache locality.

The tree has $\lceil \log_f(n) \rceil n + (\lceil \log_f(n) \rceil - 1) nf/k$ elements. Since the tree height is $\lceil \log_f(n) \rceil$, a larger f leads to an exponentially smaller number of elements. At the same time, a larger fanout requires additional storage for the pointers between tree levels and thereby linearly increases memory consumption. A larger sampling parameter k reduces memory consumption linearly. Based on the empirical evaluation in Section 6.6, we use $f = k = 32$.

5.2 Parallelization

Our implementation is completely parallelized. Embarrassingly parallel preprocessing steps use simple `parallel_for` loops. For parallel sorting during preprocessing, each task sorts a part of the data using introsort, the sorted runs are then merged. For parallel merging, we select percentiles across all sorted input runs using a median of medians and search those percentiles in each input run, thereby dividing the input runs into chunks which can be independently merged in parallel [18]. The merge sort tree is constructed bottom-up. On the lower layers, each task merges multiple input runs into an output run independently of other tasks. On the upper layers, multiple threads collaborate on merging a run, and we parallelize the merge step itself. Computing the aggregate results from the merge sort tree is again embarrassingly parallel.

5.3 Code Reuse

In contrast to the alternatives, an advantage of our approach is that we can reuse large amounts of code which exist in database systems. We reused large parts of our parallel sorting code which also handles sorting for all other `ORDER BY` clauses, e.g., for query results and non-holistic window operators. The existing sorting code was directly reusable, though needed some small fixes: Previously, the quicksort part of our introsort implementation used 2-way partitioning instead of 3-way partitioning. In the presence of duplicates, a quicksort with 2-way partitioning deteriorates to $O(n^2)$, and this quadratic behavior was triggered by framed distinct counts on columns with few duplicates because in this case most array entries are 0. Hence, we improved our quicksort implementation to use 3-way partitioning. Implementing holistic aggregates thereby improved our robustness also for other usages of sorting.

5.4 Code Generation

As mentioned, we implemented merge sort trees in a code generating database system. For code-generating systems, there is a tradeoff which parts of the algorithm to generate at runtime and which parts to implement in C++. In general, an implementation in C++ is preferable because C++ code is easier to reason about and saves query compilation time and thereby query latency. On the other hand, generated code is custom-fitted to the exact query at hand and exploits additional performance optimizations such as inlining the query-specific comparator into the sort algorithm. As such, the decision which parts to generate ad hoc and which parts to implement in C++ is crucial for overall performance.

Because the merge sort tree only stores integers independent of the query at hand, we implement the merge sort tree in C++. The preprocessing steps – which depend on the sort order specified in the query – are generated ad hoc. Thereby, merge sort trees reach both goals: We take advantage of a particular query's sort order by inlining the comparators for the preprocessing phase. At the same time, we keep implementation complexity and query compilation times low by implementing the merge sort tree in C++.

5.5 Competitors

For the evaluation, we also implemented both the naive and the incremental algorithms from Wesley and Xu [38] in our database. As discussed in Section 3.2, incremental algorithms suffer from $O(n^2)$ behavior under task-based parallelization, though. Hyper employs task-based parallelism with task sizes of 20 000 tuples. We are able to directly observe this detrimental effect on incremental algorithms in Section 6.4. For our evaluation, we still parallelized the algorithms, given that a non-parallel version of the incremental algorithms would have been completely uncompetitive.

To benchmark order statistic trees, we wrote a standalone implementation for windowed percentiles. Our implementation uses an open-source implementation of order statistic B-Trees [35].

6 EVALUATION

In this section, we experimentally evaluate our proposed algorithms against the current state-of-the-art algorithms. Given that merge sort trees beat the competing algorithms in their algorithmic complexity, we can show arbitrarily large speed-ups. Doing so would not add much insight, and hence we instead focus on the cut-over points and tradeoffs between the different algorithms.

Our benchmark results for Hyper can be reproduced through the freely available HyperAPI. All benchmark scripts for both Hyper and its competitors are available under https://github.com/salesforce/holistic-aggregates-paper.

6.1 Experimental Setup

We evaluated all algorithms on the TPC-H data set. We chose the TPC-H data set because it resembles real-world data sets and is widely available, making it easier to reproduce our results. In the spirit of the TPC-H benchmark, we modeled our queries after real-world business questions. At the same time, we kept the queries intentionally simple (no joins, no filters, ...) to keep them focused on the algorithms proposed in this paper. We executed our queries against the `lineitem` table, as this is the largest table in TPC-H.

All experiments were performed on a dual-socket system with two Intel Xeon Silver 4114 CPUs, 20 cores (40 hardware threads) at 2.2 GHz and 3.0 GHz boost frequency, running Linux 4.15 and the "performance" CPU governor. All reported query times are end-to-end times, including SQL parsing, query optimization and execution. Building the merge sort tree happens during query execution and is thereby included in the reported times. Loading the `lineitem` table from CSV is not included in our measurements. We do not build any indices during load time.

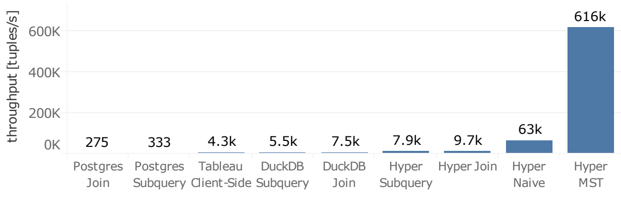

Figure 9: Throughput of a framed median on a tiny data set. Traditional SQL formulations are not competitive.

6.2 Necessity of Native Support

To show the need for the proposed SQL extensions, and the inadequacy of existing SQL functionality, we benchmark the query

```
select percentile_disc(
    0.5 order by l_extendedprice
  ) over (order by l_shipdate
    rows between 999 preceding and current row)
from lineitem
```

against a traditional SQL formulation using a correlated subquery

```
with lineitem_rn as (select *,
  ROW_NUMBER() over (order by l_shipdate) as rn
  from lineitem)
select (
  select percentile_disc(0.5)
    within group (order by l_extendedprice)
  from lineitem_rn l2
  where l2.rn between l1.rn - 999 and l1.rn)
from lineitem_rn l1
```

and a self join

```
with lineitem_rn as (. . . )
select percentile_disc(0.5)
  within group (order by l2.l_extendedprice)
from lineitem_rn l1 join lineitem_rn l2
  on l2.rn between l1.rn - 999 and l1.rn
group by l1.rn
```

We executed both formulations on DuckDB 0.3.2.dev1201, Hyper 0.0.14567 and PostgreSQL 13.4. We measured the client-side implementation in Tableau Server 2021.3 by mimicking the query with Tableau's `WINDOW_PERCENTILE` function and taking the time of the `end-tablecalc-interpreter.compute-final-table` event reported in the log file. Note that this evaluation method excludes the transfer costs between the database system and Tableau. The queries were executed on a tiny set of 20 000 tuples from the TPC-H `lineitem` table because the traditional SQL queries did not finish on larger inputs in reasonable time.

Figure 9 shows the results. The performance of the traditional SQL queries varies by an order of magnitude between database systems. In the light of this large variability, Tableau's choice to use a client-side implementation instead of offloading median computations to the database is understandable: Tableau's client-side implementation avoids the bad performance of, e.g., PostgreSQL. At the same time, it cannot reach the performance of DuckDB or Hyper. With the proposed extensions, this tradeoff changes.

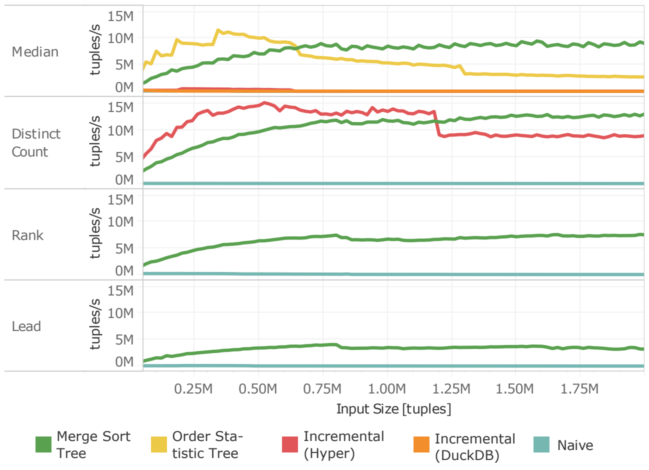

Figure 10: Throughput of holistic functions for increasing problem sizes. Not all approaches support all aggregates.

Even the naive evaluation algorithm is 15× faster than Tableau's client-side implementation and 3× faster than the fastest SQL result. This shows that, also without more efficient algorithms, the improved expressiveness provided by our SQL extensions already outperforms heavily optimized database systems like DuckDB and Hyper. Our newly proposed merge sort tree algorithm increases the performance advantage to 63× compared to the SQL results. This already gives a taste, but does not yet showcase the full performance benefits unlocked by merge sort trees. For larger input sizes, the performance differences only become larger: The non-equi join and correlated sub-query are executed as a $O(n^2)$ nested-loop joins by all competitors. While the naive evaluation algorithm avoids the overhead of a self join, it is still $O(n^2)$. The more advanced merge sort tree has a better complexity of $O(n \log n)$ but cannot fully profit from this, yet, at small input sizes. Furthermore, Hyper is single-threaded at such a small input size and does not operate at peak performance, yet. To compare the different algorithms for holistic aggregates and understand their scalability to multiple cores, we need to switch to a larger data set.

6.3 The Need for Better Holistic Algorithms

Having established the need for native holistic window function support in general, we next focus on the comparison of the available algorithms for holistic aggregates. To do so, we look at their performance at larger problem sizes. Figure 10 compares the throughput of different algorithms for increasing problem sizes. In this experiment, we use an increasingly large random sample from the TPC-H `lineitem` table as input and set the frame size to 5% of the input table size. For all order-based window functions (median, `rank`, `lead`) we ordered by `l_extendedprice`. For the distinct count, we computed the number of distinct `l_partkeys`.

We evaluated our merge sort tree based algorithms against the incremental and naive algorithms from Wesley and Xu [38] and order statistic trees [17]. For Wesley's incremental algorithm we benchmarked both an implementation in Hyper and in DuckDB. The naive algorithm was only benchmarked in Hyper. Note that

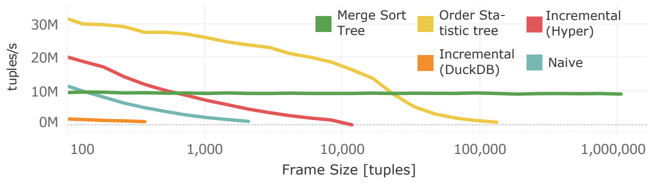

Figure 11: Throughput of a framed median for increasing frame sizes.

the order statistic tree is a standalone implementation. It thereby has an unfair advantage against the other systems which inevitably have additional overhead such as parsing and query planning.

For medians, the naive and incremental algorithms are not competitive for any problem size and never reach a throughput better than 0.6M tuples per second. They are hard to distinguish in the plot because they are overlapping and close to zero. Both the order statistic tree and the merge sort tree initially improve their throughput as input size increases. This is because at small input sizes there is not enough work for all CPU cores, yet. As stated previously, we cut tasks of 20 000 tuples and are running on a 20 core machine (40 hardware threads). We need 0.8M tuples before we have sufficient work to occupy all hardware threads. The figure confirms this and shows the merge sort tree reaching its full performance at 0.8M input tuples. While the order statistic tree is initially competitive, its performance starts to degrade at 0.35M input tuples. This is because the frame size is getting close to the task size, and the repetitive work between tasks becomes a bottleneck. By the time the merge sort tree reaches its peak performance of 9.5M tuples, the order statistic tree already fell behind the merge sort tree.

For distinct counts, the naive algorithm stays below 0.5M tuples per second, leaving the incremental algorithm as the only challenging competitor to the merge sort tree. The incremental algorithm is less affected by the repeated work across threads than the order statistic tree because re-building a hash-table is cheaper than re-building a tree. While the incremental algorithm would eventually lose against the merge sort tree due to this repeated work, in the graph we can observe a different effect: At an input size of 1.2M, the hash table maintained by the incremental algorithm outgrows the L2 cache, and the CPU drops from 0.37 instructions per cycle to 0.27 instructions per cycle due to L2 cache stalls. The merge sort tree on the other hand has a more cache-friendly memory access patterns and stays at 1.28 instructions per cycle.

For the remaining measured window functions, our only competitor is the naive algorithm which does not stand a chance. We do not show results for additional functions (other distinct aggregates, `nth_value`, `cume_dist`, ...) because those functions use merge sort trees in the same way as the shown functions with only slight modification, leading to the same performance characteristics.

6.4 Influence of Frame Sizes

Next, we measure the influence of the frame size on the different algorithms. For small window frames, even a naive evaluation strategy can yield good performance. This is because the window size directly influences the serial-execution costs of the naive and incremental algorithms and of the ordered statistic tree. Furthermore, the frame size affects their parallelizability (see Section 3.2). In this experiment, we computed the median of `l_extendedprice` on the `lineitem` table at scale factor 1 with the window frame

```
over(order by l_shipdate
  rows between size preceding and current row)
```

and measured the throughput for increasing frame sizes. Figure 11 plots the results of this experiment.

We observe that merge sort trees provide a stable performance independent of the frame size. DuckDB's implementation of the incremental algorithm is not competitive, mostly because it is not properly multi-threaded. In contrast, our implementation of both the naive and the incremental algorithm are multi-threaded and are initially competitive. This confirms our decision to multi-thread the competing algorithms, although they cannot be parallelized efficiently under task-based parallelism. As the frame size increases, the naive and incremental algorithm's throughput drops quickly, the intersection points are at 130 and 700 tuples, respectively. This is still far from the task-size of 20 000 and their bad performance is not due to bad parallelizability, but due to their worse asymptotic single-threaded complexity for increasing frame sizes. Order statistic trees are competitive for a larger range, but lose against merge sort trees at a frame size of 20 000 due to the bad effect of larger frame sizes on their parallelizability. If we do not parallelize the order statistic tree, its throughput is independent of the frame size but it is not competitive against the multi-threaded merge sort tree.

We also executed this experiment for all other window functions. The observations were similar: As we increase the frame size, the throughput of naive and incremental algorithms degrade quickly while the merge sort tree is mostly unaffected. The incremental algorithm for distinct counts loses against the merge sort tree at a frame size of 50 000, again due to the detrimental effect of parallelization at large frame sizes. Beyond that point, merge sort trees outperform their competitors by arbitrarily large factors.

This intersection point is still far from the size of SQL's default frame `BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` which has a size of 6 million tuples for this data set. For SQL's default frame, only merge sort trees compute the median in a reasonable time, with an unchanged throughput of 9.3M tuples per second.

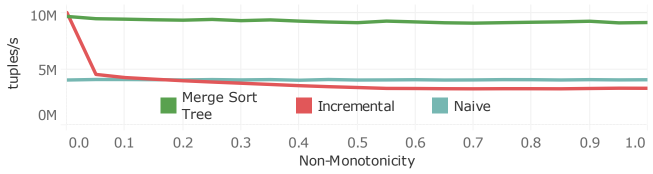

Figure 12: Throughput of a framed median for increasingly non-monotonic window frames.

6.5 Non-monotonic Frames

So far, all experiments used monotonic window frames. The incremental algorithms benefited from the large overlap between consecutive frames. To evaluate the influence of non-monotonic frames, we computed a framed median for the window clause

```
order by l_shipdate rows between
  m * mod(l_extendedprice * 7703, 499) preceding and
  500 - m * mod(l_extendedprice * 7703, 499) following
```

	Sampled Fraction of Cascading Pointers										
Fanout	1	2	4	8	16	**32★**	64	128	256	512	1024
2	1.98	1.94	1.83	1.85	1.94	1.98	2.05	2.19	2.28	2.41	2.47
4	1.47	1.40	1.34	1.33	1.34	1.39	1.45	1.55	1.66	1.83	1.89
8	1.48	1.34	1.30	1.25	1.25	1.32	1.33	1.42	1.54	1.69	1.83
16	1.49	1.28	1.23	1.24	1.25	1.29	1.34	1.44	1.51	1.66	1.82
32★	1.80	1.45	1.30	1.30	1.48	1.45	1.48	1.60	1.74	1.94	2.06
64	7.73	4.54	2.92	2.13	1.84	1.85	1.68	1.78	1.96	2.27	2.48
128	4.67	2.98	2.20	1.81	1.75	1.91	2.23	2.12	2.28	2.48	2.87
256	30.41	13.98	7.99	4.93	3.53	2.78	2.89	3.40	2.80	3.03	3.53

Figure 13: Runtime of a windowed rank for different fanout and pointer sampling parameters.

The mod generates a pseudorandom number between 0 and 499. We reused this construction from Wesley and Xu [38]. For $m = 0$, the frame is monotonic. For $m = 1$, the frames still have a size of 500 elements, but there is significantly less overlap between consecutive frames due to the added offset. As discussed in the previous experiments, at very small frame sizes the incremental algorithms can be competitive against merge sort trees. We chose such a small size to show that, even for tiny frame sizes, a non-monotonic window frame makes merge sort trees the better choice.

Figure 12 depicts the results. As expected from the previous experiment, the incremental algorithm is competitive against the merge sort tree for monotonic frames of size 500. The incremental algorithm becomes less efficient than merge sort trees as soon as we introduce a small amount of non-monotonicity. As non-monotonicity increases, the incremental algorithms become even slower than the naive implementation due to the added bookkeeping overhead. We do not depict the other window functions here for brevity, the findings are similar.

6.6 Fanout and Pointer Sampling

As stated in Section 5.1, we used the parameters $f = k = 32$. To determine those parameters, we benchmarked a single-threaded merge sort tree outside the window operator for a rank query on 1 million equally distributed random integers. Figure 13 visualizes the summed-up times for building the merge sort tree and computing the aggregate results. While $f = 16, k = 4$ provides a better execution time, we chose $f = k = 32$ because this configuration has a lower memory usage. We especially prefer a larger fanout as this leads to an exponentially reduced memory size for larger problems (see discussion in Section 5.1). With $f = 16, k = 4$, a merge sort tree on 100 million elements requires 12.4 GB, with $f = k = 32$ only 4.4 GB. The window operator and query framework needs 1.6 GB memory on this input size, without taking any aggregate state into account. Even if our competitors were completely stateless, the additional memory overhead of merge sort trees is a factor of 2.75. We deem this memory consumption acceptable given the vastly improved performance.

6.7 Cost Breakdown

Figure 14 provides a break-down of the evaluation of a running distinct count on the TPC-H `lineitem` table at scale-factor 10. All in all, query evaluation takes 3.3 seconds. The first three phases

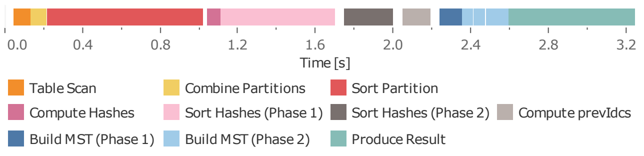

Figure 14: Execution Phases of a Framed Distinct Count.

set-up the partitioning and sorting for the window operator. The next four phases correspond to Algorithm 1: We first populate an array (line 4) which we then sort (line 5) before we compute the `prevIdcs` array (lines 7 and following). To make the sorting step independent of the data types used in the query, we do not sort the values themselves but only their hashes. In the absence of hash collisions, this does not deteriorate the runtime. Sorting is split into two phases for multi-threading: we first sort thread-locally and then merge those sorted runs. Next, we build the 6 layers of the merge sort tree, using two different multi-threading strategies for the lower and upper layers (see Section 5.2). Last, we use the merge sort tree to compute the results. All phases are parallelized. Empty spaces between phases are synchronization overheads.

7 CONCLUSION

We argue that existing SQL functionality like ranks, percentiles and distinct counts should be supported also in combination with SQL's framing capabilities for window functions. In this paper we present $O(n \log n)$ algorithms for windowed versions of all window and aggregate functions from the SQL:2011 standard except for `DENSE_RANK`. Furthermore, we show how to evaluate `DISTINCT` variants of arbitrary distributive aggregates in combination with window frames. In contrast to previous work, our algorithms are parallelized and avoid quadratic runtime in all cases, also for complex window frames including non-monotonic and non-continuous window frames.

We achieve this by mapping distinct counts, ranks, percentiles and all other window functions to range counting queries which can be answered efficiently using *merge sort trees.* While our paper focuses on relational systems, it will be interesting to see how future work can expand this approach, e.g., to stream aggregation systems where additional challenges, such as out-of-order arrivals, are present.

By implementing and evaluating the proposed algorithms in Hyper, a commercial high-performance database system, we demonstrate how the algorithmic improvements directly translate to end-to-end improvements in query time: Our implementation outperforms the best traditional SQL formulation by 63× for small inputs and our algorithm is even more performant at larger input sizes. This is both due to a better asymptotic complexity and because it can exploit multi-core systems more effectively.

REFERENCES

[1] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-Window Aggregates. In *PVLDB*. Morgan Kaufmann, 336–347. https://doi.org/10.1016/B978-012088469-8.50032-2
[2] DuckDB authors. 2021. *DuckDB - Window functions.* Retrieved June 15, 2021 from https://duckdb.org/docs/sql/window_functions

[3] PostgreSQL authors. 2021. *PostgreSQL: Documentation: 14: 3.5. Window Functions*. Retrieved January 12, 2022 from https://www.postgresql.org/docs/14/tutorial-window.html

[4] PostgreSQL authors. 2022. *PostgreSQL 14.1: gram.y source code*. Retrieved January 12, 2022 from https://github.com/postgres/postgres/blob/REL_14_1/src/backend/parser/gram.y

[5] SQLite authors. 2021. *Window functions*. Retrieved June 15, 2021 from https://www.sqlite.org/windowfunctions.html

[6] Jon Louis Bentley. 1979. Decomposable Searching Problems. *Inform. Process. Lett.* 8, 5 (1979), 244–251. https://doi.org/10.1016/0020-0190(79)90117-0

[7] Jon Louis Bentley. 1980. Multidimensional Divide-and-Conquer. *Commun. ACM* 23, 4 (1980), 214–229. https://doi.org/10.1145/358841.358850

[8] Guy E. Blelloch. 1990. *Prefix sums and their applications*. Technical Report. Carnegie Mellon University.

[9] Gerth Stølting Brodal, Beat Gfeller, Allan Grønlund Jørgensen, and Peter Sanders. 2011. Towards optimal range medians. *Theoretical Computer Science* 412, 24 (2011), 2588–2601. https://doi.org/10.1016/j.tcs.2010.05.003

[10] Gerth Stølting Brodal and Allan Grønlund Jørgensen. 2009. Data Structures for Range Median Queries. In *Proceedings of the ISAAC (Lecture Notes in Computer Science, Vol. 5878)*. Springer, 822–831. https://doi.org/10.1007/978-3-642-10631-6_83

[11] Yu Cao, Chee-Yong Chan, Jie Li, and Kian-Lee Tan. 2012. Optimization of Analytic Window Functions. In *PVLDB*. 1244–1255. https://doi.org/10.14778/2350229.2350243

[12] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. In *Proceedings of the CIKM*. ACM, 1201–1210. https://doi.org/10.1145/2983323.2983807

[13] Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. 2014. Linear-Space Data Structures for Range Mode Query in Arrays. *Theory of Computing Systems* 55, 4 (2014), 719–741. https://doi.org/10.1007/s00224-013-9455-2

[14] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica* 1, 2 (1986), 133–162. https://doi.org/10.1007/BF01840440

[15] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional Cascading: II. Applications. *Algorithmica* 1, 2 (1986), 163–191. https://doi.org/10.1007/BF01840441

[16] 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. In *PVLDB*. 1313–1324. https://doi.org/10.14778/1454159.1454171

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[18] Rhys S. Francis, Ian D. Mathieson, and Linda Pannan. 1993. A Fast, Simple Algorithm to Balance a Parallel Multiway Merge. In *Proceedings of PARLE (Lecture Notes in Computer Science, Vol. 694)*. Springer, 570–581. https://doi.org/10.1007/3-540-56891-3_46

[19] Sariel Har-Peled and S. Muthukrishnan. 2008. Range Medians. In *Proceedings of the European Symposium on Algorithms*. Springer, 503–514. https://doi.org/10.1007/978-3-540-87744-8_42

[20] Joseph M Hellerstein and Michael Stonebraker. 2005. *Readings in database systems, 4th edition*. MIT press.

[21] Snowflake Inc. 2021. *Window Functions — Snowflake Documentation*. Retrieved June 15, 2021 from https://docs.snowflake.com/en/sql-reference/functions-analytic.html

[22] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *SIGMOD*. ACM, 281–293. https://doi.org/10.1145/2882903.2882957

[23] Allan Grønlund Jørgensen and Kasper Green Larsen. 2011. Range Selection and Median: Tight Cell Probe Lower Bounds and Adaptive Data Structures. In *Proceedings of the SODA*. ACM-SIAM, 805–813. https://doi.org/10.1137/1.9781611973082.63

[24] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *SIGMOD*. ACM, 1001–1013. https://doi.org/10.1145/3448016.3457288

[25] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. 2003. Range Mode and Range Median Queries on Lists and Trees. In *Proceedings of the ISAAC (Lecture Notes in Computer Science, Vol. 2906)*. Springer, 517–526. https://doi.org/10.1007/978-3-540-24587-2_53

[26] 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*. ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[27] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. In *PVLDB*. 1058–1069. https://doi.org/10.14778/2794367.2794375

[28] Microsoft. 2021. *OVER Clause (Transact-SQL) - SQL Server | Microsoft Docs*. Retrieved June 15, 2021 from https://docs.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql

[29] Oracle. 2021. *Analytic Functions*. Retrieved June 15, 2021 from https://docs.oracle.com/cd/E11882_01/server.112/e41084/functions004.htm

[30] Anatoli U. Shein, Panos K. Chrysanthis, and Alexandros Labrinidis. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. In *Proceedings of the EDBT*. OpenProceedings.org, 397–408. https://doi.org/10.5441/002/edbt.2018.35

[31] Ivan Smirnov. 2016. *Is it possible to query number of distinct integers in a range in O(lg N)?* Retrieved September 11, 2021 from https://stackoverflow.com/a/39797537/3043948

[32] Tableau. 2021. *Table Calculation Functions*. Retrieved June 15, 2021 from https://help.tableau.com/current/pro/desktop/en-us/functions_functions_tablecalculation.htm

[33] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2019. Optimal and General Out-of-Order Sliding-Window Aggregation. In *PVLDB*. 1167–1180. https://doi.org/10.14778/3339490.3339499

[34] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-Window Aggregation. In *PVLDB*. 702–713. https://doi.org/10.14778/2752939.2752940

[35] Simon Tatham. 2017. *Counted B-Trees*. Retrieved January 21, 2022 from https://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html

[36] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2021. Scotty: General and Efficient Open-source Window Aggregation for Stream Processing Systems. *ACM Trans. Database Syst.* 46, 1 (2021), 1:1–1:46. https://doi.org/10.1145/3433675

[37] 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 *Proceedings of the DBTest Workshop at SIGMOD*. ACM, 1–6. https://doi.org/10.1145/3209950.3209952

[38] Richard Wesley and Fei Xu. 2016. Incremental Computation of Common Windowed Holistic Aggregates. In *PVLDB*. 1221–1232. https://doi.org/10.14778/2994509.2994537