Incremental Computation of Common Windowed Holistic Aggregates

Richard Wesley
Tableau Software
837 North 34th Street
Seattle, WA 98103
hawkfish@tableau.com

Fei Xu
Tableau Software
837 North 34th Street
Seattle, WA 98103
fxu@tableau.com

ABSTRACT

Windowed aggregates are a SQL 2003 feature for computing aggregates in moving windows. Common examples include cumulative sums, local maxima and moving quantiles. With the advent over the last few years of easy-to-use data analytics tools, these functions are becoming widely used by more and more analysts, but some aggregates (such as local maxima) are much easier to compute than others (such as moving quantiles). Nevertheless, aggregates that are more difficult to compute, like quantile and mode (or "most frequent") provide more appropriate statistical summaries in the common situation when a distribution is not Gaussian and are an essential part of a data analysis toolkit.

Recent work has described highly efficient windowed implementations of the most common aggregate function categories, including *distributive*[1] aggregates such as cumulative sums and *algebraic* aggregates such as moving averages. But little has been published on either the implementation or the performance of the more complex *holistic* windowed aggregates such as moving quantiles.

This paper provides the first in-depth study of how to efficiently implement the three most common holistic windowed aggregates (count distinct, mode and quantile) by reusing the aggregate state between consecutive frames. Our measurements show that these incremental algorithms generally achieve improvements of about 10× over naïve implementations, and that they can effectively detect when to reset the internal state during extreme frame variation.

1. INTRODUCTION

Window functions (also known as analytic OLAP functions) are defined by the SQL:2003 standard and implemented to varying degrees by a number of database systems ([1, 2, 3, 4, 5, 16, 19, 6]). Moreover, in addition to these back end database systems, a number of user–facing data analysis tools ([7, 8]) provide analogues for many of these SQL functions as part of their analytic environment. SQL interface tools such as SQLShare [17] also provide window functions in response to significant demand for them in scientific data analytics.

[1]These terms are defined in Section 2.2

Proceedings of the VLDB Endowment, Vol. 9, No. 12
Copyright 2016 VLDB Endowment 2150-8097/16/08.

Window functions can be used to perform a number of analytically useful calculations such as rankings, cumulative sums and moving statistical summaries. While this functionality can often be implemented in earlier versions of SQL, such implementations are extremely complex.

Traditional windowed aggregates (such as moving averages) are well suited to Gaussian data distributions where the average is the same as the median and the mode, but real world data sets often present more interesting distributions where the median and mode are not the same as the average. To track how such distributions move over time, an analyst needs access to more appropriate statistical summaries, such as modes and quantiles.

Implementations of windowed versions of such aggregates are almost non-existent. We were only able to verify implementations for windowed `median` in Oracle and HANA. Oracle also has limited support for windowed `quantile` and `count distinct`, but only implemented partitioning, and the aggregate was computed over the entire partition. Moving windowed `quantile` and `count distinct` are not supported. No database or research system in our survey had support for windowed `mode`.

As a result, we have implemented three of the most important *holistic* aggregates: `count distinct`, `mode` and discrete `quantile`. These aggregates are problematic to implement efficiently as simple aggregates, and windowed versions are even more challenging. Previous work on efficient windowed aggregation done by Lies *et al.* [20] focussed on the traditional aggregates. Our work builds on theirs by adding windowed implementations of these more difficult aggregates. We show that careful reuse of traditional aggregation data structures in the windowing environment can lead to significant performance gains over simply calling the database's existing aggregate function in each window. Specifically our contributions include:

- Multiple algorithms for efficiently implementing the most common holistic aggregates;
- Rules for choosing between these algorithms;
- Measurements showing a range of performance gains ranging from 1.5x to nearly 200x over the naïve implementations.

The rest of the paper is organised as follows. Section 2 provides some detailed background on window functions terminology. Sections 3 through 5 provide details of the algorithms for `count distinct`, `mode` and discrete `quantile`. Our experimental results are presented in Section 6. We then discuss related work in Section 7 and conclude in Section 8.

2. BACKGROUND

Window functions were introduced in SQL 2003 to accommodate complex analytical tasks, because such computations are typically extremely difficult if not impossible to implement in older versions of SQL. When it is possible, the implementations often involve features such as recursive queries and self-joins, and are quite verbose.

In order to appreciate the complexity of windowed aggregation functions, we need to first describe the computation environment of different classes of SQL functions, including window functions. We also need to understand the complexity of different aggregation types because we will be discussing windowed aggregates of the most complex type.

2.1 Computation Environments

SQL functions can be grouped by their computation environment into the following three classes:

- *Tuple* functions, whose computation environment is a single tuple. These functions combine existing attributes in the current tuple to compute a new attribute in the same tuple. Examples include arithmetic operations such as `(A+B)`, and data manipulation functions such as `year(ShipDate)`.

- *Aggregate* functions, whose computation environment is a set of tuples defined by a common grouping key. These functions combine multiple values of a single attribute that share a common set of values for the grouping attributes. Examples include `sum(Sales) ... group by State, City` and `median(Delay) ... group by Sensor`.

- *Window* functions, whose computation environment is a set of adjacent tuples. These functions compute a new attribute for each row by combining values from neighboring rows, where "neighboring" is defined by a partitioning, an ordering and a frame. Examples include `rank()` **over (partition by Sport order by Score desc)** and `avg(Sales)` **over (partition by Team order by Date rows between 3 preceding and 3 following)**.

The environment for Window functions is by far the most complex and consists of three levels as shown in Figure 1.

- The tuples being processed are first *partitioned* into disjoint groups based on 0 or more partition keys.

- The tuples in each partition are then *ordered* by sorting the partition on 0 or more sorting keys.

- Finally, each window function can specify a subset of the partition called a *frame* that restricts the computation to a consecutive set of rows within the ordered partition.

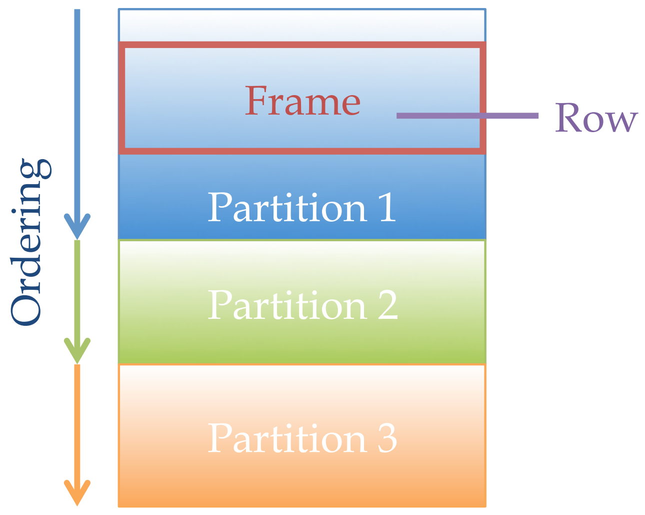

Figure 1: The Window Computation Environment

Frames are specified using a number of tuples preceding or following the tuple whose value is being computed. In addition to a number of tuples, frame edges can also be unbounded. When both edges are unbounded, then the frame degenerates into the entire partition.

As a concrete example of a windowed aggregate, consider the computation given above by `avg(Sales)` **over (partition by Team order by Date rows between 3 preceding and 3 following)**. This function computes a moving average of sales data for each team over time. Partitioning by team keeps different teams' sales data separate. Ordering by date specifies the time series for the moving average. The `rows preceding/following` clause defines the aggregation frame and restricts the moving average to a seven-day window.

Frames can also be specified using ranges of values relative to the current value of the (single) ordering column. Range frames can be converted to row frames by using either forward scans (for fixed frames) or by using binary search (for variable frames). Framing mode is independent of the type of aggregate being computed, so we will assume from this point on that we are using row framing.

Not all window functions specify frames, and aggregates in particular can be either framed or unframed. Nevertheless, we can implement unframed aggregates by simply using the framed aggregate with a frame specification that is unbounded at both ends. Unframed aggregates produce only one value for the entire frame, and as optimising this is trivial, we will assume from this point on that all aggregates are using bounded framing.

2.2 Aggregate Complexity

Aggregate functions have traditionally [14] been grouped into three categories based on their complexity:

- *Distributive* aggregates are those whose computation can be "distributed" and recombined. Common examples of distributive aggregates include `sum` and `max`.

- *Algebraic* aggregates are those whose computation is a simple algebraic function of the data and other aggregates. Common examples of algebraic aggregates include `average` and `variance`.

- *Holistic* aggregates are those whose computation requires looking at all the data at once, and hence their evaluation cannot be decomposed into smaller pieces. Common examples of holistic aggregates include `count distinct` and `median`.

Holistic aggregates are the most difficult to compute efficiently because they cannot be decomposed into smaller pieces. All of the aggregates we consider here (*i.e.* `count distinct`, `mode` and `quantile`) are holistic.

2.3 Previous Work

Window functions are a relatively new feature of SQL, but research on their optimisation is growing. The first research publication on how to optimise window function queries is given by Cao *et al.* [9]. Their work examines how to optimise multiple window functions with different window definitions. They propose multiple physical `Window` operators based on the partitioning and sorting state of the previous window, including the traditional Full Sort and two new operators called Hashed Sort and Segmented Sort. Our work is based on improving the performance of the Hashed Sort operator.

The most relevant previous work is the implementation of a `Window` operator proposed by Leis *et al.* [20], and implemented in HyPer [18], an in-memory database system. They provided implementations of a number of important window functions using a block–style interface, where multiple rows are processed in a single call. They also present multiple algorithms (such as Removable Cumulative Aggregation) and acceleration structures (such as Segment Trees) for computing windowed aggregates. However, the aggregate window functions they discuss are limited to distributive and algebraic aggregates such as `sum` and `average`. The goal of our work is to extend the class of windowed aggregates to include the most common holistic aggregates, and to show how to compute them relatively efficiently.

The HyPer `Window` operator is similar to the Hashed Sort operator from [9] and is highly scalable. Their implementation first partitions the windowed table by hashing the partition keys into a fixed number (1024) of bins and then sorts each bin on both the partition and sort keys. We have based our `Window` operator on this model.

2.4 Our Implementation

We implemented our `Window` operator in the Tableau Data Engine (TDE) [23], a block-iterated column store used in the Tableau product suite. The TDE uses a traditional Volcano [13] operator tree for execution. The physical operator family consists of materialising (stop-and-go) operators called *Tables* and non-materialising (iterating) operators called *Flows*. The `Window` operator was implemented as a Table operator that consumes another Table operator as input.

During the hash partitioning phase of `Window`, we use 256 hash buckets because we materialise the column containing the hash values and this choice allows us to materialise a column that is only one byte wide, which reduces the memory overhead by a factor of two. This value is at the low end of the bucket count "sweet spot" from Figure 9 in [20], and our evaluation in Section 6.2 shows that this choice does not impact performance.

Frame	Data	Result
0	[3 4 3 2] 7 2 5 4	3
1	3 [4 3 2 7] 2 5 3	4
2	3 4 [3 2 7 2] 5 3	3
3	3 4 3 [2 7 2 5] 3	3
4	3 4 3 2 [7 2 5 3]	4

Table 1: Count Distinct Example

The block–iterated interfaces used in the TDE are similar to the interfaces used by HyPer, and this design makes it easy to share state between consecutive frames for incremental evaluation of windowed aggregates.

For historical reasons, our sort code is a standard `introsort` [21] with parallel sub-sorts, which has been adapted to sort pairs of columns. We call out the implications of this in the discussion in Section 6.

3. WINDOWED COUNT DISTINCT

The first holistic aggregate we will investigate is `count distinct`. Consider the following SQL query which uses this aggregate:

```
select State, count(distinct Department)
from Sales
group by State
```

This query returns the number of unique departments that made sales in each state. States that have a low number may need to have more marketing resources devoted to them to improve sales.

We could implement this functionality in terms of simpler SQL operations by using a nested query:

```
select State, count(*)
from (select State, Department
    from Sales
    group by State, Department)
group by State
```

This two-level implementation works well for a single use of `count distinct`, but if there is more than one (*e.g.* we added a `count distinct` of customers to provide context), then each count will have to be computed separately using a variant of this two-level query and then explicitly joined back to the main query on State:

```
select State, DeptCount, CustCount
from(
        select State group by State as S
        left join (
        select State, count(*) as DeptCount
        from (select State, Department
                from Sales
                group by State, Department)
        group by State ) as D
        on (S.State = D.State)
        ) left join (
        select State, count(*) as CustCount
        from (select State, Customer
                from Sales
                group by State, Customer)
        group by State ) as C
        on S.State = C.State
        )
```

Algorithm 1 Naïve Count Distinct

```
 1: procedure NaiveCountFrame($counts$, $values$, $F$)
 2:     $counts \leftarrow \{\}$
 3:     for $f \in F$ do
 4:         $counts[values[f]] \mathrel{+}= 1$
 5:     end for
 6: end procedure
 7:
 8: procedure NaiveCountD($result$, $values$, $frames$)
 9:     $counts \leftarrow \{\}$
10:     for $i \in [0, |result|)$ do
11:         $F \leftarrow frames[i]$
12:         NaiveCountFrame($counts$, $values$, $F$)
13:         $result[i] \leftarrow |counts|$
14:     end for
15: end procedure
```

Algorithm 2 Incremental Count Distinct

```
 1: procedure Increment($nonzero$, $counts$, $value$)
 2:     if $value \in counts$ then
 3:         $counts[value] \mathrel{+}= 1$
 4:     else
 5:         $counts[value] = 1$
 6:         $nonzero \mathrel{+}= 1$
 7:     end if
 8: end procedure
 9:
10: procedure Decrement($nonzero$, $counts$, $value$)
11:     $counts[value] \mathrel{-}= 1$
12:     if $counts[value] = 0$ then
13:         $nonzero \mathrel{-}= 1$
14:     end if
15: end procedure
16:
17: procedure IncrementalCountD($result$, $values$,
        $frames$)
18:     $P \leftarrow [0, 0)$
19:     $counts \leftarrow \{\}$
20:     $nonzero \leftarrow 0$
21:     for $i \in [0, |result|)$ do
22:         $F \leftarrow frames[i]$
23:         if $nonzero \leq \tau \cdot |counts|$ then
24:             NaiveCountFrame($counts$, $values$, $F$)
25:             $nonzero \leftarrow |counts|$
26:         else
27:             for $f \in F \setminus P$ do
28:                 Increment($nonzero$, $counts$, $values[f]$)
29:             end for
30:             for $f \in P \setminus F$ do
31:                 Decrement($nonzero$, $counts$, $values[f]$)

32:             end for
33:         end if
34:         $result[i] \leftarrow nonzero$
35:         $P \leftarrow F$
36:     end for
37: end procedure
```

This approach generates two hash tables per aggregate (one for the group by and one for the join) and rapidly becomes unwieldy. Because of this complexity, `count distinct` is sometimes implemented as an aggregate function that uses a single hash set per row as the aggregate state. The hash set tracks the unique elements of the domain that have been seen and the final result is simply the size of the hash set. This approach only requires one hash table per aggregate, reducing complexity and increasing performance. It also is much easier to implement in a windowing context because we do not need to create aggregates and joins for every row in the output.

3.1 Naïve Count Distinct

The simplest windowed algorithm for `count distinct` is to use a hash set as in the aggregate function state implementation and clear it at the start of each frame. We call this algorithm *Naïve Count Distinct* and it is shown as Algorithm 1.

In our implementation, we try to reuse the main hash set memory, but the content buckets are freed for each frame.

3.2 Incremental Count Distinct

To compute the next `count distinct` value, we may be able to reuse the hash table from the previous frame. Consider the example in Table 1 with a 4-element wide frame. There are 3 distinct values in the brackets of Frame 0.

When we move the frame one element to the right to Frame 1, we remove a 3 on the left but there is still one remaining, and we also added a new value (7) on the right, so the new count of distinct values is 4. The second 3 will remain in the window for one more frame before being removed.

We can model this incremental change by tracking not only the values, but their counts using a hash *map*. Whenever we move to the next frame, we then only have to update the hash map counts for the values that are removed and those that are added. We call this algorithm *Incremental Count Distinct* and it is shown as Algorithm 2.

In the example, we have used a single frame that moved by one element each time, but frames can be variable width and the changes between consecutive frames can be more than a single element at each end. Nevertheless, each frame consists of a single interval F, so two consecutive frames P and F are either disjoint, or there are two (possibly empty) intervals that belong either to $F \setminus P$ or $P \setminus F$. If the consecutive frames are disjoint, then we can fall back to the naïve algorithm, but if not we can increment the counts for the added frame values in $P \setminus F$ at Line 28 and then decrement the counts for removed frame values in $F \setminus P$ at Line 31. The result is then the number of non-zero counts.

3.3 Autocorrelation

Suppose now that we have just moved to Frame 3, where the second 3 has just been removed. In this situation, we can either delete the hash table bucket or leave it with a count of 0. If the domain being aggregated is uncorrelated with the ordering dimension, such as counting distinct ZIP codes for a time window in a data warehouse, then we want to retain buckets that become empty because we are likely to use them again soon. In our example, we can see that 3 is about to be reused in Frame 4, so freeing and then reallocating the bucket a short time later is going to reduce performance.

Conversely, if we are counting a domain that is highly correlated with the ordering dimension, such as counting ship dates over that same time window, then we want to delete

Frame	Data	Result	Count
0	[*c d c b*] *g b e d*	c	2
1	*c* [*d c b g*] *b e d*	(g)	1
2	*c d* [*c b g b*] *e d*	b	2
3	*c d c* [*b g b e*] *d*	b	2
4	*c d c b* [*g b e d*]	(d)	1

Table 2: Mode Example

buckets that become empty because they are wasting space in the hash table. In our example, this would correspond to having the numbers tending to increase over time.

We adaptively detect these two scenarios during the windowing operation by comparing the number of empty buckets to the total size at Line 23 and flushing the hash table when the ratio reaches a threshold τ. Note that setting $\tau \leftarrow 1$ reduces the algorithm to Naïve Count Distinct. In Section 6.3 we determine empirically that 0.25 is a good value for τ.

However, τ is an imperfect parameter because it cannot distinguish between true autocorrelation and a low frame size to domain size ratio. If the window size is smaller than the domain size (*e.g.* 10 and 100 respectively) then $\tau > 0.1$ would trigger a flush, even though the hash table will never grow larger than 100 entries. This is impossible to determine without knowing the domain size *a priori*, because that domain size is effectively what we are calculating. Our measurements in Section 6.3 show, however, that even with this imperfect rule, the performance with flushing is better than always clearing. Future work is needed in this area, and one simple heuristic is to not flush hash tables up to a certain size *e.g.* half of the L1 cache.

3.4 Frame Reordering

It is tempting to consider whether one could reorder the frames in such a manner as to provide maximum overlap between frame computations, but determining this reordering is NP-Hard (see the Appendix A for details.) A heuristic approach to frame reordering may be a promising area for future work, but it is beyond the scope of this paper.

4. WINDOWED MODE

The next aggregate we shall consider is `mode`. Recall that the *mode* of a distribution is the most common value and consider the following SQL query:

```
select State, mode(Department)
from Sales
group by State
```

This query returns the Department with the highest transaction volume for each State. The result could be used to bootstrap a system where departments with expertise in particular states can be enlisted to help other departments improve their sales through consultation with the local experts.

Implementing a single `mode` aggregate by using elementary SQL constructs is quite complex and involves multiple self-joins (Appendix B). The implementation presented there uses five hash tables to compute the mode value, and would require a sixth to join back to the fact table were other aggregates requested.

Due to this complexity, it is generally preferable to implement `mode` by using an aggregate function with internal state. This state is similar to that for `count distinct` but a hash map from the values to the counts is maintained for each aggregate state, together with the maximum count and the corresponding value. The final result is the value corresponding to the largest count seen so far.

Algorithm 3 Naïve Mode

```
1: procedure ModeAdd($mode$, $nonzero$, $counts$, $value$)
2:     $counts[value] \mathrel{+}= 1$
3:     $count \leftarrow counts[value]$
4:     if $count = 1$ then
5:         $nonzero \leftarrow nonzero + 1$
6:     end if
7:     if $count >= mode.count$ then
8:         $mode.valid \leftarrow true$
9:         $mode.value \leftarrow value$
10:        $mode.count \leftarrow count$
11:    end if
12: end procedure
13:
14: procedure ModeRm($mode$, $nonzero$, $counts$, $value$)
15:    $count \leftarrow counts[value]$
16:    if $count = 1$ then
17:        $nonzero \leftarrow nonzero - 1$
18:    end if
19:    $counts[value] \mathrel{-}= 1$
20:    if $count == mode.count$ && $value = mode.value$ then
21:        $mode.valid \leftarrow false$
22:    end if
23: end procedure
24:
25: procedure NaiveMode($result$, $values$, $frames$)
26:    for $i \in [0, |result|)$ do
27:        $F \leftarrow frames[i]$
28:        $mode.count \leftarrow 0$
29:        $counts \leftarrow \{\}$
30:        $nz \leftarrow 0$
31:        for $f \in F$ do
32:            ModeAdd($mode$, $nz$, $counts$, $values[f]$)
33:        end for
34:        $result[i] \leftarrow mode.value$
35:    end for
36: end procedure
```

4.1 Naïve Mode

The simplest windowed algorithm for `mode` is to use a hash map containing counts (as in the aggregate function state implementation) and clear the map for each frame. We call this algorithm *Naïve Mode* and it is shown as Algorithm 3. In our implementation, we reuse the main hash table memory, but the content buckets are freed each time. Note that we update the *mode* at Line 7 even when the *count*s are equal because we do not check the *valid* flag. This functionality will be used in the next section.

4.2 Incremental Mode

When we move to the next `mode` frame, we may be able to reuse the hash map from the previous frame, along with the previous result value and its count. To illustrate the process, we use the example data from Table 2

At Frame 0, we have determined that the `mode` is c with a count of 2. When we move to Frame 1, we lose one of the c values, so we no longer have a valid `mode`. Moreover, any new value that gets added cannot be guaranteed to be a valid `mode` unless its count exceeds that of the previous `mode`. Since g is a new value that has never been seen before, it only has a count of 1 so we cannot assume that it is the new `mode`. When we have processed all the new values and not found a new valid `mode`, we then have to rescan the hash map to find a new mode with a smaller count. Since all the counts are 1, we simply choose the last one that we see (g).

When we move to Frame 2, we remove d (which is not the current `mode`) and then add a new value b that is already in the hash map with a count of 1. This is greater than the count for the previous `mode`, so we know that we have a new valid `mode` without having to rescan the hash map.

We call this algorithm *Incremental Mode* and present it as Algorithm 4.

Algorithm 4 Incremental Mode

```
 1: procedure IncrementalMode($result$, $values$,
        $frames$)
 2:     $counts \leftarrow \{\}$
 3:     $nz \leftarrow 0$
 4:     $P \leftarrow [0, 0)$
 5:     for $i \in [0, |result|)$ do
 6:         $F \leftarrow frames[i]$
 7:         if $nz \leq \tau \cdot |counts|$ then
 8:             $counts \leftarrow \{\}$
 9:             $nz \leftarrow 0$
10:             $mode.count \leftarrow 0$
11:             for $f \in F$ do
12:                 ModeAdd($mode$, $nz$, $counts$, $values[f]$)
13:             end for
14:         else
15:             for $f \in P \setminus F$ do
16:                 ModeRm($mode$, $nz$, $counts$, $values[f]$)
17:             end for
18:             for $f \in F \setminus P$ do
19:                 ModeAdd($mode$, $nz$, $counts$, $values[f]$)
20:             end for
21:         end if
22:         if not $mode.valid$ then
23:             $mode.count \leftarrow 0$
24:             for $c \in counts$ do
25:                 if $c.count > mode.count$ then
26:                     $mode.valid \leftarrow true$
27:                     $mode.value \leftarrow c.value$
28:                     $mode.count \leftarrow c.count$
29:                 end if
30:             end for
31:         end if
32:         $result[i] \leftarrow mode.value$
33:         $P \leftarrow F$
34:     end for
35: end procedure
```

The algorithm first goes through the subrange of the previous frame that lies outside the current frame ($P \setminus F$) at Line 15 and decrements the *count*s of those values (possibly to zero). If the *count*s before decrementing and the *value*s are the same, then we have "lost track" of the real *mode* because we no longer know whether there is another *value* in the hash table that also had the same *count*. In this situation, we mark the *mode* as invalid at Line 21, but we retain the *count* in case we exceed it in the next step.

Once we have removed all the values from $P \setminus F$, we start adding in new values from $F \setminus P$ at Line 18. If this process ever generates an updated *count* for a *value* that exceeds the *count* determined by the previous frame, then this is a new valid *mode* candidate, and we can start tracking again as in the naïve algorithm. If we never generate such a new *mode* candidate, then take a full pass over the hash table at Line 24 to find the new *mode*.

Just as with Incremental Count Distinct, we have the choice between deleting or keeping buckets with counts of 0 when there is autocorrelation. To detect this, we again track the number of non-zero buckets and use τ to determine the flushing threshold. In Section 6.3 we determine empirically that 0.25 is again a good value for τ.

4.3 Comparison to Count Distinct

The incremental versions of both `count distinct` and `mode` use the same data structure (a hash map from value to count) but they differ in one important respect: While `count distinct` can always produce its result after the hash map update process, `mode` may need to be re-derived if the old `mode` is removed from the frame and the new `mode` is not created by the new additions to the frame. In theory, this re-derivation can reduce the performance of `mode` relative to `count distinct` by a large amount. Consider the case where only one value is changed in a large hash table, but that value happens to be the previous mode. The entire hash table would then have to be re-scanned to generate the new mode, which results in quadratic behaviour. In our measurements, we only found a drop off of about 10%, but future work may be able to quantify this more accurately.

5. WINDOWED QUANTILES

The two aggregates we have just discussed are quite similar in their implementation, but we move now to continuous and discrete `quantile`, which require a significantly different approach.

The most common approach for computing `quantile` aggregates is to use a variant of Floyd and Rivest's QuickSelect algorithm [12] for finding the kth element in a list of N elements. This is algorithm is $O(N)$ in the number of elements being selected from, and has the side effect of partially sorting the list, so that all values less than the selected value end up to the left of it and all values greater end up to the right of it. For reference, we have provided `qselect` and its utility function `partition` as Algorithm 6 and Algorithm 5 respectively. Note that the versions given here include indirection to avoid modifying the *values* array.

5.1 Discrete Quantiles

The *discrete* `quantile` aggregate function has the signature $quantile(expression, fraction)$ where $fraction$ is a real number varying between 0 and 1. It computes the value closest to a given fractional position in the naturally ordered values of the expression (including duplicates.) For example, given a set of expression values $(0, 0, 2, 3, 4, 5, 6, 7, 8, 8, 10)$ and a fraction of 0.2, the corresponding quantile value is 2. Setting $fraction$ equal to 0.5 implements the discrete `median` aggregate (which has the value 5 in the example).

Algorithm 5 Partition

```
1: function PARTITION($values$, $index$, $l$, $r$, $p$)
2:     $v \leftarrow values[index[p]]$
3:     $index[p] \leftrightarrow index[r]$
4:     $s \leftarrow l$
5:     for $i = l$, $i < r$, $i \leftarrow i + 1$ do
6:         if $values[index[i]] < v$ then
7:             $index[s] \leftrightarrow index[i]$
8:             $s \leftarrow s + 1$
9:         end if
10:    end for
11:    $index[s] \leftrightarrow index[r]$
12:    return $p$
13: end function
```

Algorithm 6 QuickSelect

```
1: procedure QSELECT($k$, $values$, $index$, $l$, $r$)
2:     while $l < r$ do
3:         select $p \in [l, r)$
4:         $p \leftarrow$ PARTITION($values$, $index$, $l$, $r - 1$, $p$)
5:         if $k = p$ then break
6:         else if $k < p$ then
7:             $r \leftarrow p$
8:         else
9:             $l \leftarrow p + 1$
10:        end if
11:    end while
12: end procedure
```

Discrete `quantile` can be easily implemented using QuickSelect by converting the quantile value into the index k.

5.2 Continuous Quantiles

It is often the case, however, that the quantile dividing point is not an integral index into the list of values. The simplest example happens when computing the median of an even number of values. For categorical domains (like strings) we have to choose a value on one side or the other of the "true" quantile, but for continuous domains (such as real numbers) we have the option of averaging the two values on either side of the true value. This average is called the *continuous* `quantile` function. Working from the example above, if we were to remove the last value from the list, then the median would fall between 4 and 5 and the continuous `quantile` would be 4.5.

From a computational point of view, however, when the result of the continuous `quantile` differs from the discrete `quantile` (because the quantile element falls between two other elements), it suffices to perform two discrete `quantile` computations and average the results. Moreover, we can search for the first candidate k such that the other element k' would be in the smaller of the intervals $[0, k)$ and $(k, w-1]$. Selecting k' from its interval consists of searching for the *max* (resp. *min*) value in that interval.

Searching for a simple extreme like this is twice as fast as QuickSelect (there is no iteration), and assuming that the cost of the few arithmetic instructions used in the averaging operation is negligible, we can see that computing a continuous quantile is $O(w) + \frac{1}{2}O(w/2)$ or roughly a factor of 1.25. Thus the implementation of continuous quantile can be expressed in time-bounded terms of the discrete solution, and without loss of generality, we can restrict our attention to the discrete case for the rest of this discussion.

Algorithm 7 Naïve Quantile

```
1: procedure NAIVEQUICKSELECT($result$, $values$,
       $quants$, $frames$)
2:     for $i \in [0, |result|)$ do
3:         $F \leftarrow frames[i]$
4:         $k \leftarrow \lfloor quants[i] * (|F| - 1) \rfloor$
5:         $index \leftarrow []$
6:         $j \leftarrow 0$
7:         for $f \in F$ do
8:             $index[j] \leftarrow f$
9:             $j \leftarrow j + 1$
10:        end for
11:        QSELECT($k$, $values$, $index$, 0, $j$);
12:        $result[i] \leftarrow values[index[k]]$
13:    end for
14: end procedure
```

Algorithm 8 Quantile with Reuse

```
1: procedure REUSEQUANTILE($result$, $values$,
       $quants$, $frames$)
2:     $P \leftarrow [0, 0)$
3:     $index \leftarrow []$
4:     for $i \in [0, |result|)$ do
5:         $F \leftarrow frames[i]$
6:         $k \leftarrow \lfloor quants[i] * (|F| - 1) \rfloor$
7:         REUSEINDEXES($index$, $F$, $P$)
8:         QSELECT($k$, $values$, $index$, 0, $j$)
9:         $result[i] \leftarrow values[index[k]]$
10:        $P \leftarrow F$
11:    end for
12: end procedure
```

5.3 Naïve Quantile

For some applications the reordering caused by QuickSelect may be acceptable, but if we use it to compute a windowing aggregate, we need to restore the data to its original order before we can compute the next value. We can avoid this problem by sorting pointers (or indexes) to the values instead of the values themselves.

The simplest implementation of `quantile` then is to build an array of indexes for each window and then compute the kth element using QuickSelect. We call this algorithm *Naïve Quantile* and it is shown as Algorithm 7. In our implementation, we reuse the index array memory but not its contents.

Naïve Quantile is quadratic in the size of the frame and the size of the partition, so we now investigate how reusing the index could improve performance.

5.4 Quantile with Reuse

When we come to compute the next `quantile` value, we don't have to rebuild the index array from scratch. Instead, we can simply scan the array, replacing any indexes that have fallen out of the frame and inserting the new ones in the holes. We call this algorithm *Quantile with Reuse* and show it as Algorithm 8, but the bulk of the reuse logic is in a utility function shown as Algorithm 9.

Algorithm 9 Quantile Index Reuse

```
 1: procedure ReuseIndexes($index$, $F$, $P$)
 2:     $j \leftarrow 0$
 3:     for $f \in [0, |P|)$ do
 4:         $idx \leftarrow index[f]$
 5:         if $j \neq f$ then
 6:             $index[j] \leftarrow idx$
 7:         end if
 8:         if $idx \in F$ then
 9:             $j \leftarrow j + 1$
10:         end if
11:     end for
12:     if $j > 0$ then
13:         for $f \in F \setminus P$ do
14:             $index[j] \leftarrow f$
15:             $j \leftarrow j + 1$
16:         end for
17:     else
18:         for $f \in F[j:]$ do
19:             $index[j] \leftarrow f$
20:             $j \leftarrow j + 1$
21:         end for
22:     end if
23: end procedure
```

The scan takes place at Line 3 and only starts shifting indexes down at Line 6 when it has previously encountered an index at Line 8 that is not in the new frame F. Once the old frame indexes from P have been shifted down, we can insert the new indexes from F at Line 13. If we did not find any overlap, then we fall back to the NaïveQuickSelect algorithm at Line 18

For highly variable frame sizes it may sometimes be worth starting over when the intersection between the two frames is small, but for frames that vary little, we can avoid large numbers of cache writes by preserving as much of the existing index set as possible.

Another advantage of index reuse is that the values are already partially ordered [15] when we move to the next frame position. This in turn leads to better sampling of the data for partitioning and fewer reordering operations.

In summary, *Quantile with Reuse* works best when there is significant overlap between consecutive frames. The ability to reuse not only the indexes but their partial ordering reduces the amount of work. This work is still $O(W)$, but the constant can be significantly smaller.

5.5 Quantile with Replacement

In the fixed frame case, we typically replace only *one* value as the window moves right one position. Fixed frames are by far the most common frame type because varying the size of a windowed statistical summary has limited utility. In this situation it is extremely cheap to verify whether we need to re-select at all. We call this algorithm *Quantile with Replacement* and show it as Algorithm 11, but the main logic is again in a utility function, this one shown as Algorithm 10.

The main algorithm checks for fixed frames at Line 8. If it finds one, it uses the *ReplaceIndex* function. This first scans for the new hole and breaks at Line 7 when it finds one. It then checks whether the inserted value sorts before (*resp.* after) the previous quantile and is inserted before (*resp.* after) it at Line 14. If either test succeeds, then the previous

Algorithm 10 Quantile Index Replacement

```
 1: function ReplaceIndex($k$, $values$, $index$, $F$, $P$)
 2:     $same \leftarrow false$
 3:     $j \leftarrow 0$
 4:     for $f \in [0, |P|)$ do
 5:         $idx \leftarrow index[f]$
 6:         if $j \neq f$ then
 7:             break
 8:         end if
 9:         if $idx \in F$ then
10:             $j \leftarrow j + 1$
11:         end if
12:     end for
13:     $index[j] \leftarrow F[-1] - 1$
14:     if $k < j$ then
15:         $same \leftarrow result[i-1] < values[index[j]]$
16:     else if $j < k$ then
17:         $same \leftarrow values[index[j]] < result[i-1]$
18:     end if
19:     return $same$
20: end function
```

Algorithm 11 Quantile with Replacement

```
 1: procedure ReplaceQuantile($result$, $values$,
        $quants$, $frames$)
 2:     $P \leftarrow [0, 0)$
 3:     $index \leftarrow []$
 4:     for $i \in [0, |result|)$ do
 5:         $F \leftarrow frames[i]$
 6:         $k \leftarrow \lfloor quants[i] * (|F| - 1) \rfloor$
 7:         $same \leftarrow false$
 8:         if $|F| = |P|$ && $P[0] + 1 = F[0]$ then
 9:             $same \leftarrow$ ReplaceIndex($k$, $values$, $index$,
    $F$, $P$)
10:         else
11:             ReuseIndexes($index$, $F$, $P$)
12:         end if
13:         if $same$ then
14:             $result[i] \leftarrow result[i-1]$
15:         else
16:             qselect($k$, $values$, $index$, 0, $|F|$)
17:             $result[i] \leftarrow values[index[k]]$
18:         end if
19:         $P \leftarrow F$
20:     end for
21: end procedure
```

value is reusable. In the other cases, we have to re-select, but because the data is already close to correct, the number of operations is greatly reduced.

The primary benefit of *Quantile with Replacement* is that it can avoid re-selecting roughly 50% of the time. Since that is the most expensive step in the algorithm, avoiding it has measurable performance benefits. In Section 6.5, we show that this estimate is borne out in practice for larger fixed frames with a the Replace algorithm showing a 2× performance gain over Reuse.

6. EVALUATION

The algorithms proposed in this paper for count distinct, mode and quantile have been integrated into Tableau Data

Engine. In this section, we experimentally evaluate our implementations. We attempted to compare our implementations with state-of-the-art commercial and open-source database systems, but unfortunately, most systems do not implement any windowed holistic aggregates. For the rest that do implement some windowed holistic aggregates, we experimented with them and found that the implementation was incomplete (*e.g.* only partition keys with unbounded frames were supported.) Therefore, it is not feasible to compare our implementation with state-of-the-art commercial or open-source database systems. In this section, we first compare our implementation framework with the HyPer implementation in [20] for non-holistic aggregates, and use the result as the baseline to show the performance of our implementations. The rest of the experiments focus exclusively on a comparative performance study of our proposed algorithms.

6.1 Experimental Setup

The experiments were performed with an Intel Xeon dual-processor E5-2630 v2 with 12 cores (24 hardware threads) at 2.6 GHz. The system has 32 GB of RAM and three SSD drives. We used Windows 7 Enterprise SP1 as an operating system and Microsoft Visual C++ 2013 as a compiler.

All our experiments were performed using 2-column 64-bit integer tables of 10M rows with varying definitions for the two columns for defining different partitioning (column `a`) and ordering (column `b`) scenarios.

- The `rank1` data set has a constant partition value, and ordering values that are ascending, dense and unique. The table is sorted on a hash of the ordering column.
- The `rank100` data set has uniformly distributed set of 100 partition values, and ordering values that are ascending, dense and unique. The table is initially sorted on the ordering column itself.
- The `rank10M` data set has two columns of ascending, dense and unique values, but with different minimum values. The table is initially sorted on the two columns.

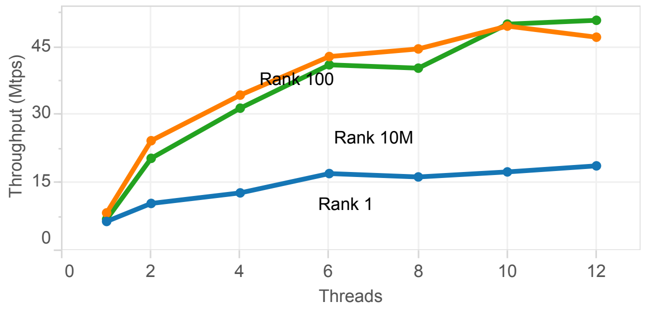

Figure 2: Scalability of `rank` queries

6.2 Window Operator Performance

In this section, we benchmark our implementation using the experimental setup from [20] to demonstrate that we correctly implemented their prior work on regular window functions and windowed distributive aggregates. After this validation, will proceed to an evaluation of our newer holistic aggregates. The results were generally similar, so we restrict our presentation to two representative experiments.

The first benchmark measures the scalability of our `rank` implementation against multiple cores on the following query:

```
select rank()
   over (partition by a order by b desc)
from rankP
```

The results are shown in Figure 2. Throughput measurements are given in *Million tuples per second* (Mtps).

The resulting throughput is as expected (approximately 45 Mtps as in [20]), verifying the correctness of our implementation of the `Window` operator itself. The performance of the `rank1` data set appears to be bottlenecked by our columnar sorting, which is most apparent when we have a single partition.

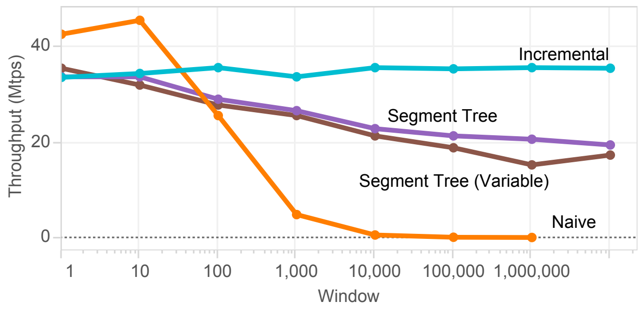

Figure 3: Performance of `sum` queries with constant frame bounds for different frame sizes

The second benchmark measures the scalability of the `sum` framed aggregate function using four algorithms with different frame sizes. For this benchmark and the remaining experiments, we use the query:

```
select agg(a) over (order by b asc rows
    between W−1 preceding and current row )
from rank100
```

where *agg* is replaced by the aggregate under consideration. The results are shown in Figure 3. The Naïve algorithm uses a fixed width trailing frame. The Incremental algorithm is a single value replacement algorithm, again with a fixed trailing frame. The Segment Tree algorithm uses precomputed segment trees as in [20] with a fixed trailing frame. The Segment Tree (Variable) algorithm uses precomputed segment trees and a fixed size frame, but we perform a simple pseudo-random integer computation to vary the frame boundaries around each value:

```
select agg(a) over (
  order by b asc
  rows between mod( b * p1, p2 ) preceding
  and W − mod( b * p1, p2 ) following )
from rank100
```

The primes ($p1$ and $p2$) are chosen based on the frame size, with $p2$ being roughly half of W and $p1$ being much larger. The performance difference between the fixed and variable segment tree runs is due to the frame computation overhead.

As expected, the Incremental and Segment Tree algorithms have performance that is roughly linear in the size of the frame with throughput around 25Mtps [20].

Having established our baseline performance, we turn now to the evaluation of our new algorithms.

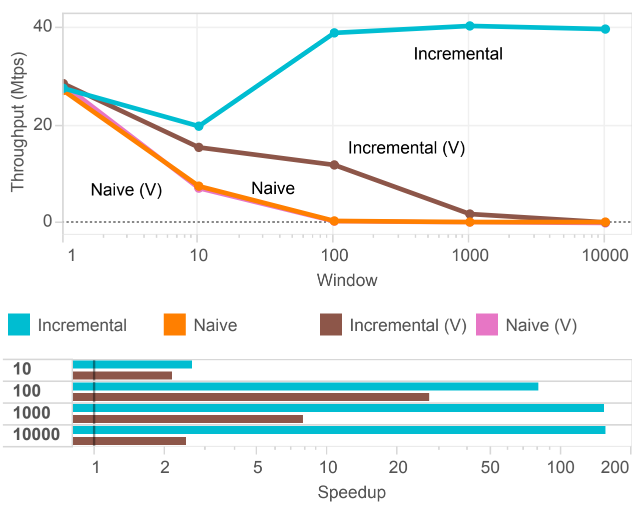

Figure 4: Performance of `count distinct` queries

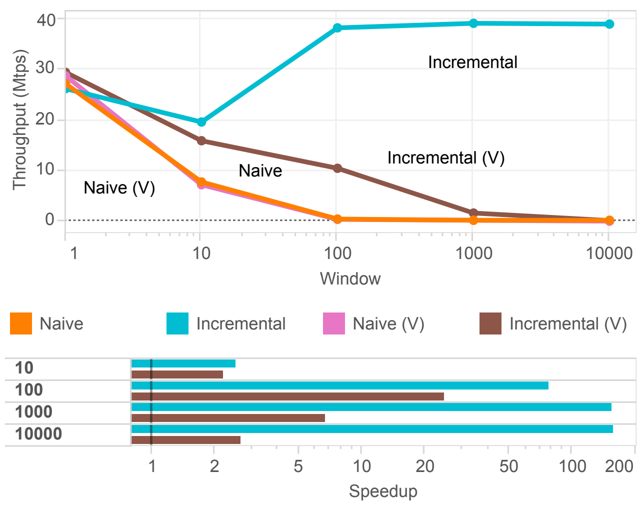

Figure 5: Performance of `mode` queries

6.3 Count Distinct and Mode Evaluation

We compared the Naïve and Incremental algorithms for both `count distinct` and `mode` by running them at both fixed width trailing frames and pseudo-random frame starts. The results are presented in Figure 4 and Figure 5 (note that we have used *log* axes for Speedup here due to the quadratic nature of the improvements.) With the high locality of the fixed width trailing frames, we see little impact of the window size upon throughput, and for pseudo-random frame starts, we obtain significant performance gains over the equivalent naïve measurements.

It is interesting to note that, even though only Incremental Count Distinct can guarantee that the hash map will not be rescanned, in practice Incremental Mode seems to provide similar performance gains.

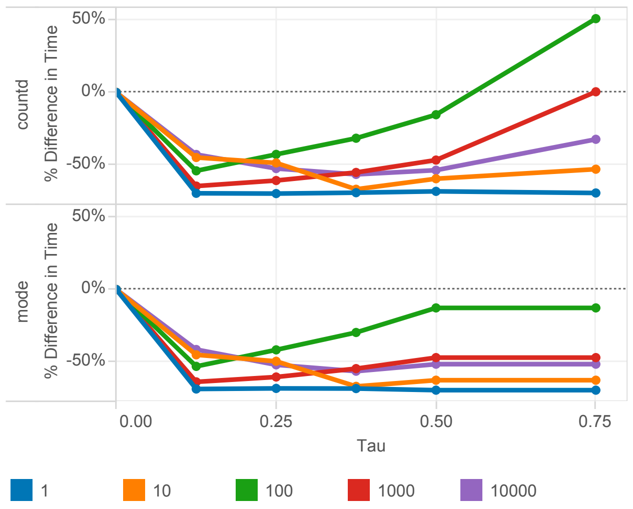

Figure 6: Performance change with varying τ

6.4 Determination of Tau

We also performed some experiments to determine reasonable values for τ (the hash table flushing threshold) in both hash table algorithms. Because τ is driven by the level of correlation between the window ordering and value being counted, we used the unique ordering field b in the following query:

```
select agg(b) over (order by b asc rows
    between W−1 preceding and current row )
from rank100
```

The relative performance of various frame sizes against τ are shown in Figure 6. Values of τ greater than 0.0 and less than 0.5 provide balanced trade-off between different frame sizes. Values approaching 1.0 lead to highly degraded quadratic performance with this data and were not measured. We have chosen 0.25 for our experiments.

The dip in Incremental performance at frame size 10 in Figures 4 and 5 is caused by hash table flushing because the ratio of the frame size to the domain size (100 in this case) is lower than our value of τ. The performance gain over the Naïve algorithm is more than 2×, which seems an acceptable trade-off. Improving this heuristic is left for future work.

6.5 Quantile Evaluation

There are three different `quantile` algorithms to evaluate (Naïve, Reuse and Replace) and we have profiled them with fixed width trailing frames. We also profiled the Reuse and Naive algorithms with pseudo-random frame starts, to remove the effects of the frame computation. Because the expected number of comparisons for QuickSelect is highest for a quantile value of 0.5, we have elected to measure `median` to cover the worst case. The results are shown in Figure 7.

All runs show a drop off as the frame size increases, but the fixed frame Replace algorithm consistently outperforms Naïve, and even increases its lead as the frame size increases, exceeding 10× when the frame size reaches 10K elements. The Reuse algorithm also improves its performance gains over the Naïve versions with increasing fixed frame size, but

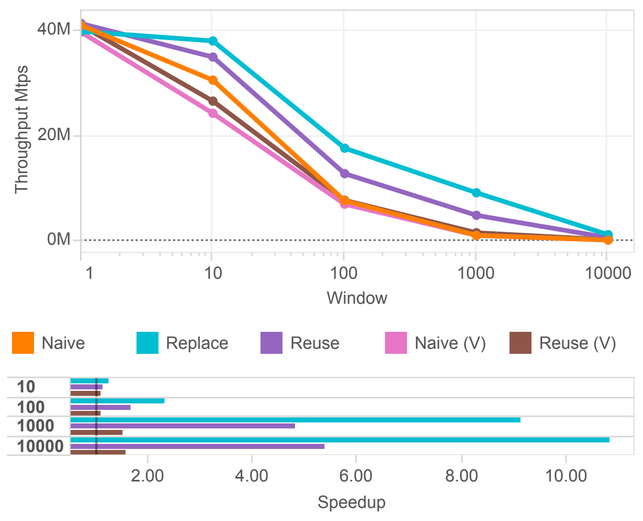

Figure 7: Performance of `median` queries

only by about 6× at the 10K level. Even the Reuse algorithm with a wildly varying frame start gives an improvement of 1.5× at large frame sizes over the corresponding Naïve algorithm.

7. RELATED WORK

Window functions were first defined as an optional part of the SQL:1999 standard before being fully incorporated into SQL:2003 [25]. By 2011, all major commercial and open source databases had provided implementations, including Oracle [1], IBM DB2 [3], Microsoft SQL Server [2], SAP HANA [4] and Postgres [5]. User facing analytic packages have also added similar functionality, in that time period, including Tableau (2010) [7] and Spotfire [8].

In recent years, the body of work on window operators and their associated functions has been growing. Cao *et al.* [9] discuss optimising execution of multiple window operators through careful ordering derived from the partitioning and ordering specifications. Interestingly, they also found an NP-hard problem in their work and provided a heuristic to work around this difficulty. We believe that the existence of such hard problems in this area is related to the utility of window functions for useful data analytics. The recent work of Leis *et al.* [20] described a highly efficient framework for implementing window operators and their functions, but their work only focused on distributive and algebraic aggregates. See Section 2.3 for a more detailed discussion of their work.

Aggregate functions have been part of relational queries since the original SEQUEL language proposal [10]. More recently, incremental approaches to ordered aggregates were explored by Yang and Widom [24], and to windowed aggregates by Leis *et al.* [20]. Sliding window algorithms for aggregates in data stream management systems were also studied by Zhu and Shasha [26] and Datar *et al.* [11].

The need for more complete windowed analytics in SQL is discussed in [17]. Their work followed the use of SQL by academic data scientists and found the lack of such support to be one of the key barriers to adoption of relational databases for scientific data analysis.

8. SUMMARY AND FUTURE WORK

As data analysts become more sophisticated, they are starting to expect more computational depth from their tools, and non-Gaussian statistical summaries such as quantiles are the first step beyond Gaussian summaries such as mean. Computing such measures quickly is essential for maintaining interactive analytic flow, but the holistic nature of these aggregates can make interactive performance elusive.

Holistic aggregates are difficult to compute because of the need to maintain global state for the entire aggregate. This difficulty has resulted in no complete commercial or open-source implementations of these windowed aggregates to date. We have presented several algorithms for computing three holistic windowed aggregates (count distinct, mode and quantile) that are generally faster than naïve implementations by reusing this shared state between frames. Some of the implementations are linear in the size of the data for common windowing scenarios (such as fixed–size moving window and running aggregates) and all of them are at worst quadratic with smaller constants than the naïve versions. These constant differences can range from around 1.5x to nearly 200x in our test cases.

Future work in this area could focus on improving the performance of these aggregates under extreme variations in frame size and location, such as more precise characterisation of the cost–benefits of data structure reuse. We would also like to consider more realistic data sets with different data volumes, data skews, window sizes and domain sizes. Another area for future work would be to determine an effective heuristic for frame reordering for circumventing the NP-hard nature of the overlap maximisation problem. Finally, there are other interesting statistical functions (such as the spatio-temporal `presence` measure [22]) that might benefit from our incremental approach.

9. REFERENCES

[1] `http://docs.oracle.com/database/121/DWHSG/analysis.htm`.

[2] `https://msdn.microsoft.com/en-us/library/ms189461(v=sql.120).aspx`.

[3] `http://www-01.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0023461.html`.

[4] `https://help.sap.com/hana/SAP_HANA_SQL_and_System_Views_Reference_en.pdf?original_fqdn=help.sap.de`.

[5] `http://www.postgresql.org/docs/9.4/static/tutorial-window.html`.

[6] `https://www.monetdb.org/Documentation/Manuals/SQLreference/WindowFunctions`.

[7] `http://onlinehelp.tableau.com/current/pro/online/windows/en-us/functions_functions_tablecalculation.html`.

[8] `https://docs.tibco.com/pub/spotfire/6.0.0-november-2013/userguide-webhelp/ncfe/ncfe_advanced_custom_expressions.htm`.

[9] Y. Cao, C.-Y. Chan, J. Li, and K.-L. Tan. Optimization of Analytic Window Functions. *Proc. VLDB Endow.*, 5(11):1244–1255, July 2012.

[10] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of*

the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM.

[11] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 635–644, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[12] R. W. Floyd and R. L. Rivest. Algorithm 489: The Algorithm SELECT – for Finding the Ith Smallest of N Elements [M1]. *Commun. ACM*, 18(3):173–, Mar. 1975.

[13] G. Graefe. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb. 1994.

[14] J. M. Hellerstein and M. Stonebraker. *Readings in Database Systems, 4th Edition*. MIT Press, 2005.

[15] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[16] D. Inkster, M. Zukowski, and P. Boncz. Integration of Vectorwise with Ingres. *SIGMOD Rec.*, 40(3):45–53, Nov. 2011.

[17] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. SQLShare: Results from a Multi-Year SQL–as–a–Service Experiment. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, New York, NY, USA, 2016. ACM.

[18] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[19] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[20] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann. Efficient Processing of Window Functions in Analytical SQL Queries. *Proc. VLDB Endow.*, 8(10):1058–1069, June 2015.

[21] D. R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8):983–993, Aug. 1997.

[22] S. Orlando, R. Orsini, A. Raffaetà, A. Roncato, and C. Silvestri. Trajectory data warehouses: Design and implementation issues. *JCSE*, 1(2):211–232, 2007.

[23] R. Wesley, M. Eldridge, and P. T. Terlecki. An Analytic Data Engine for Visualization in Tableau. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1185–1194, New York, NY, USA, 2011. ACM.

[24] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proceedings of the 17th International Conference on Data Engineering*, pages 51–60, Washington, DC, USA, 2001. IEEE Computer Society.

[25] F. Zemke. What's New in SQL:2011. *SIGMOD Rec.*, 41(1):67–73, Apr. 2012.

[26] Y. Zhu, D. Shasha, and Y. Z. D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *In VLDB*, pages 358–369, 2002.

APPENDIX

A. FRAME REORDERING COMPLEXITY

Let $\mathcal{F}$ be the set of all frames in a partition, and let $\mathcal{G}$ be the totally connected graph using $\mathcal{F}$ as its vertices. Define the weight of the connection between two frames F and G by $w(F, G) := |F \cap G|$. Then the total frame overlap along any path through $\mathcal{G}$ is simply the sum of the weights of the connecting nodes. In order to maximise the total overlap, we need to find the path with the maximum length. But this problem is trivially isomorphic to the Traveling Salesman problem, which is NP-Hard.

B. MODE USING BASIC SQL

```
with Counts as (
        select State, Department, count(*)
            as Mode
        from Sales
        group by State, Department )
select State, min(Department) as Mode
from (
        Sales as S
        inner join (
        select State, Department
        from (
                Counts as C
                inner join (
                select State, max(Mode) as
                    Mode
                from Counts
                ) as D
                on (C.State = D.State and C
                    .Mode = D.Mode)
        ) as M
        on S.State = M.State
group by State
```