Data Science and Engineering (2018) 3:40–51
https://doi.org/10.1007/s41019-018-0060-x

Approximate Calculation of Window Aggregate Functions via Global Random Sample

Guangxuan Song[1] · Wenwen Qu[1] · Xiaojie Liu[1] · Xiaoling Wang[1]

Received: 26 August 2017 / Revised: 19 November 2017 / Accepted: 28 February 2018 / Published online: 15 March 2018

Abstract
Window functions have been a part of the SQL standard since 2003 and have been studied extensively during the past decade. They are widely used in data analysis; almost all the current mainstream commercial databases support window functions. However, in recent years the size of datasets is growing steeply; the existing window function implementations are not efficient enough. Recently, some sampling-based algorithms (e.g., online aggregation) are proposed to deal with large and complex data in relational databases, which offer us a flexible trade-off between accuracy and efficiency. However, few sampling techniques has been considered for window functions in databases. In this paper, we extend our previous work (Song et al. in Asia-Pacific web and web-age information management joint conference on web and big data, Springer, pp 229–244, 2017) and proposed two new algorithms: range-based global sampling algorithm and row-labeled sampling algorithm. The proposed algorithms use global sampling rather than local sampling and are more efficient than other existing algorithms. And we find our proposed algorithms out performed the baseline method over the TPC-H benchmark dataset.

Keywords Window function · Query optimization · Sample

1 Introduction

Window functions, as a feature of relational database, were introduced in SQL:1999 as extended documents and formally specified in SQL:2003. Window functions allow aggregation over a set of tuples within a particular range instead of a single tuple. Unlike aggregation with the group-by clause which only outputs one tuple for each group, the OVER() clause associates each tuple in a window with the aggregated value over the window. As a result, window functions greatly facilitate the formulation of business intelligence queries by using the aggregation functions such as ranking, percentiles, moving averages and cumulative totals. Besides, comparing with grouped queries, correlated subqueries and self-joins [2, 3], window functions also improve the readability of the scripts. The syntax of a window function is so user-friendly that database users without professional programming skills can write business intelligence queries effortlessly. Due to these desirable features, window functions are widely used by current mainstream commercial databases in recent years, such as Oracle, DB2, SQL Server, Teradata, (Pivotal's) Greenplum [4–6].

However, the existing implementation strategies of window functions are too expensive to process big data. While some pioneering works are focusing on improving the time efficiency of executing window functions [7–11], the speed improvements cannot keep up with the ever-growing data size generated by modern applications, and none of them can meet the requirement of real-time processing which is vital to online services.

Researchers have been using various sampling techniques for query processing to accommodate the growing data size. Sampling are effective as analytical queries do not always require precise results. It is often more desirable to return an approximate result with an acceptable margin of error quickly than waiting for the exact calculation for a long time. Database users can specify confidence intervals

✉ Xiaoling Wang
xlwang@sei.ecnu.edu.cn

[1] International Research Center of Trustworthy Software, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China

in advance and stop the query processing as soon as desired accuracy is reached.

Online aggregation is a classic sampling technique first proposed in 1997 [12]. It enables users to terminate the processing of a query as soon as the result is acceptable. This is particularly useful for expensive analytical queries. For example, if the final answer is 1000, after k seconds, the user gets the estimates in form of a confidence interval like [990, 1020] with 95% probability. This confidence interval keeps on shrinking as the system gets more and more samples. Online aggregation has been used in standard SQL queries, such as join [13] and group by [14]. However, few systems fully support online aggregation. And best to our knowledge we are the first to study online aggregation for window functions.

This work develops several different sampling approaches, based on the sampling techniques in online aggregation, for executing window functions. The proposed sampling algorithms speed up query processing time by reducing the number of tuples used in window aggregation functions. And this work is an extension of our work [1], in which we proposed naive random sampling (NRS) and incremental random sampling (IRS) algorithms.

The main contributions of this paper are summarized as follows:

- We designed the global random sampling for fast execution of window functions with ROW mode and RANGE mode: Range-based global sampling algorithm (RGSA) and row-labeled sampling algorithm (RLSA). To improve time efficiency, our methods sample a subset from the original global dataset and then estimate the results with quality guarantee based on the subset.
- We proved that our algorithms generate unbiased estimators for various aggregate functions and provide an approach to calculate the confidence intervals. We also specified how to adjust parameters (e.g., sampling rate) according to the unique characteristics of the window functions, to satisfy various application scenarios.
- Our algorithms outperformed the native algorithm in PostgreSQL based on the TPC-H benchmark.

The rest of this paper is organized as follows. Section 2 discusses the background of window functions and sampling and introduces the related work. Section 3 describes details of native algorithm in PostgreSQL and IRS and then presents the formulae to calculate confidence intervals. Section 4 explains the differences between RANGE and ROW mode, and then we show the details of RGSA and RLSA. We show the experimental results in Sect. 5. Finally, this paper is concluded in Sect. 6 with a vision toward the future work.

2 Background

2.1 Window Functions

Window functions are part of the SQL:2003 standard in order to accommodate complex analytical queries. The general form of a window aggregate function is as follows:

```
SELECT *, Agg(expression) OVER(
  [PARTITION BY partition_list]
  [ORDER BY order_list [frame_clause]])
  FROM table_list;
```

where *Agg* can be any of the standard aggregation functions such as *SUM*, *AVG*, *COUNT* and *expression* is the attribute to be aggregated upon. The *OVER* clause defines the aggregation window, inside which *PARTITION BY*, *ORDER BY* and *frame_clause* determine the properties of the window jointly(In fact, a frame is just a window). Examples include *MAX*(velocity) *OVER* (*PARTITION BY* roadid,carid *ORDER BY* time_t *ROWS BETWEEN 10 PRECEDING AND 10 FOLLOWING*), which means we monitor the maximum speed of the car within 20 min of each time in a car monitoring application.

Figure 1 illustrates the process of determining the range of a window. Firstly, the *PARTITION BY* clause divides the tuples into disjoint groups according to the attribute columns in *partition_list*. Nextly, attributes in *order_list* are used to sort tuples in each partition. Finally, we confirm the bounds of the frame, i.e., the range of tuples to aggregate upon, for the current tuple by *frame_clause* and invoke the related aggregate function to compute the aggregation result. The first two processes are called *reordering*, and the last one is called *sequencing*.

For each row in a partition, a window frame identifies two bounds in the window and only the rows between these two bounds are included for aggregate calculation. There are two frame models, *ROWS* and *RANGE*. In the *ROWS* model, the boundaries of a frame can be expressed as how

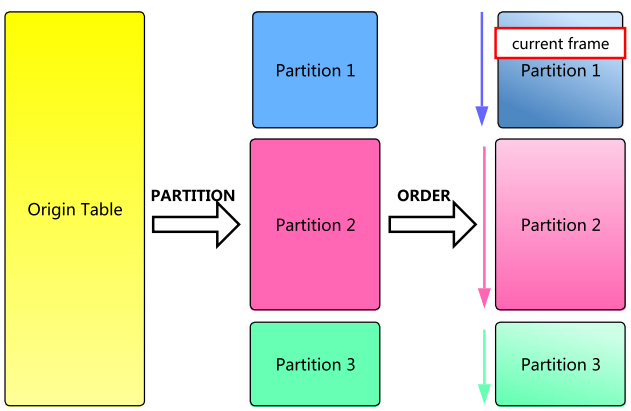

Fig. 1 Determine the window of current tuple

many rows before or after the current row belong to the frame; in the *RANGE* model, the boundaries are specified as a range of values with respect to the value in the current row [10].

Like traditional aggregate functions, a window aggregate function is composed of a state transition function *transfun* and an optional final calculation function *finalfun*. To compute the final result of an aggregate function, a transition value is maintained for the current internal state of aggregation, which is continuously updated by *transfun* for each new tuple seen. After all input tuples in the window have been processed; the final aggregate result is computed using *finalfun*. Then we determine the frame boundaries for the next tuple in the partition and do the same work again.

2.2 Sampling

Sampling has been widely used in various fields of computer science, such as statistics, model training, query processing. With the advent of Big Data era, sampling is gaining unprecedented popularity in various applications, since it is prohibitively expensive to compute an exact answer for each query, especially in an interactive system. Sampling techniques provide an efficient alternative in applications that can tolerate small errors, in which case exact evaluation becomes an overkill.

Online aggregation [12] a classic approach based on sampling provides users with continuous updates of aggregation results as the query evaluation proceeds. It is attractive since estimates can be timely returned to users, and the accuracy improves with time as the evaluation proceeds.

There are four components in a typical online aggregation system, as shown in Fig. 2. The first one is the approximate result created by the system. The second and third parts consist of reliability and confidence interval which reflect the accuracy and feasibility of the result. The progress bar is the last part that reports the schedule

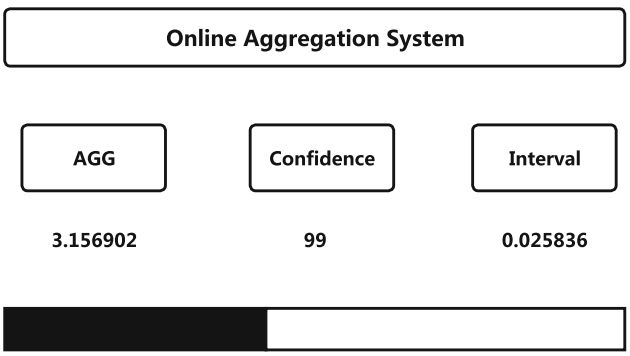

Fig. 2 An online aggregation system

information for the current task. All values improve over time until they meet a user's needs.

Since the results generated by sampling are approximate, it is important to calculate an associated confidence interval. For a simple aggregate query on one table, like *SELECT op* (*expression*) *FROM table WHERE predicate*, it's easy to achieve online aggregation by continuously sampling from the target table. Standard statistical formulas can help us get unbiased estimators and estimate the confidence interval. A lot of previous work [13–16] have made great contributions on this problem.

2.3 Related Work

A window function is evaluated in two phases: resorting and sequencing. Accordingly, existing approaches for evaluating window functions mainly fall into two categories.

Cao et al. [7] improved the full sort technique for evaluating window functions by proposing more competitive hash-sort and segment-sort algorithms. Then Cao et al. [8] proposed a novel method for the multi-sorting operations in one query. They found that most computation time is spent in sorting and partitioning when window size is small. And they proved that it is an NP-hard problem to find the optimal sequence to avoid repeated sorting for multiple window functions in one query and provided a heuristic method. Other earlier studies, such as [9, 17, 18] and [19], made a lot of contributions with regard to the ORDER BY clause and GROUP BY clause and they proposed optimization algorithms based on either the function dependencies or the reuse of the intermediate results.

Besides, there are several recent works about window operators and their functions. Leis et al. [10] proposed a general execution framework of window functions which is more suitable for memory-based systems. An novel data structure named segment tree is proposed to store aggregates for sub-ranges of an entire group, which helps reduce redundant calculations. However, the approach is only suitable for distributive and algebraic aggregates. Wesley et al. [11] proposed an incremental method for three holistic windowed aggregates. And Song et al. [20] provided an general optimization method based on shared computing, which uses temporary windows to store the shared results and only need a small amount of memory. However, all the above works are toward exact evaluation and they face performance bottlenecks when processing massive data.

Sampling has been used in approximate query evaluation [21, 22], data stream processing [23], statistical data analysis [24] and other fields [25, 26]. Moreover, as the classic application of sampling, Hellerstein et al. [12] first proposed the concept of online aggregation. Since then,

research on online aggregation has been actively pursued. Xu et al. [14] studied online aggregation with *group by* clause and Wu et al. [16] proposed a continuous sampling algorithm for online aggregation over multiple queries. Qin and Rusu [27] extended online aggregate to distributed and parallel environments. Li et al. [13] studied online aggregation on the queries with join clauses. These techniques based on sampling give us an opportunity to calculate the aggregate easily. In general, sampling is required to be uniform and independent in order to get reasonable results, but both [13] and [28] proved that a non-uniform or a non-independent samples can be used to estimate the result of an aggregate.

There are many other valuable works [29–32] on query optimization and sampling, but none of them deals with the problem of window functions. In this paper, we study how to evaluate online aggregate for queries involving and extend the existing work [1] for new application scenarios. What's more, work [33] extended the definition of window functions for graph computing and proposed three different types of graph window: unified window, K-hop window and topological window. This provides a way for us to achieve cross-domain development of the window in the future.

3 Window Function Execution

Definition 1 (*window W*) A window consists of a set of rows and is denoted by a triple $W_i(h, t, V)$, where i indicates that it is the i-th window, h is the window head and t is the window tail, and V is the transition value for the window.

3.1 Native Algorithm

As mentioned in Sect. 2, window functions in commercial database systems are generally evaluated over a windowed table in a two-phase manner. In the first phase, the windowed table is reordered into a set of physical window partitions, each of which has a distinct value of the *PARTITION BY* key and is sorted on the *ORDER BY* key. The generated window partitions are then pipelined into the second phase, where the window function is sequentially invoked for each row over its window frame within each window partition.

Algorithm 1 Native algorithm

Input: the whole table T after partitioning and reordering
Output: aggregate result for each tuple in the table

```
1:  for each partition in the table T do
2:    for each window W_i in the partition do
3:      init W_i.h,W_i.t,W_i.V;
4:      if W_i.h=W_{i-1}.h then
5:        W_i.V ← W_{i-1}.V;
6:        for each row r in (W_{i-1}.t, W_i.t] do
7:          W_i.V ← transfunc(W_i.V, r);
8:        end for
9:      else
10:       for each row r in [W_i.h, W_i.t] do
11:         W_i.V ← transfunc(W_i.V, r);
12:       end for
13:     end if
14:     return W_i.V;
15:   end for
16: end for
```

Firstly, for each window W_i in the *partition*, we initialize the parameters of W_i and place the read pointer at the start of the window (Line 3). Then determine whether the start position of the current window is equal to that of the previous one. If the condition is true, we can reuse the previous result and assign it to $W_i.V$ (Line 5). Nextly, we place the read pointer at the end of the previous window and traverse all tuples from $W_{i-1}.t$ to $W_i.t$. If it is false, we just traverse all tuples from $W_i.h$ to $W_i.t$. Finally, return the result and prepare for the next window.

An example of the processing of window functions is shown in Fig. 3, where the aggregate function is summation and the frame model adopted is *ROW*. The entire table is partitioned and sorted according to *attr*_2 and makes summation on *attr*_1. Red brackets in Fig. 3 represent windows. Then we find that *W*2 begins with the same tuple as *W*1 and the result of *W*2 is incrementally updated. However, the start tuple is different between *W*2 and *W*3, we have to traverse all the tuples to evaluate a result.

For a table with N tuples, if we use a general window function query, like *SUM*(*attr*_1) *OVER* (*PARTITION BY*

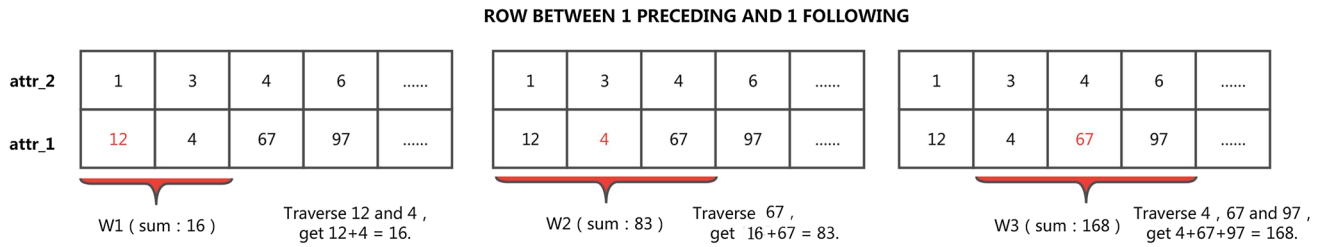

Fig. 3 An example of window functions

attr_2 *ORDER BY* attr_2 *ROWS BETWEEN S / 2 PRECEDING AND S / 2 FOLLOWING*). In most cases, we must use S tuples to get one result for each tuple. And the total execution cost is $\Theta(NS)$ without considering partitioning and sorting.

NRS is the base algorithm in paper [1], which using native random sampling in each window. Here, we only introduce the improved version named ISR in the paper. Since a large number of common tuples are shared by adjacent windows, ISR reuses the sampling results of the previous window.

Algorithm 2 Incremental Sampling

Input: the whole table T after partition and order
Output: approximate result for each tuple in the table

```
1:  for each partition in the table T do
2:    for each window W_i in the partition do
3:      initialize W_i and compute the boundaries of W_i;
4:      SW_i ← SW_{i-1};
5:      while SW_i.indexarr[SW_i.sh] < W_i.h do
6:        remove_head(SW_i);
7:      end while
8:      create an empty SW structure named T_SW;
9:      if SW_i.indexarr[SW_i.st] < W_i.t then
10:       T_SW_i ← sample(SW_i.indexarr[SW_i.st], W_i.t, rate);
11:     end if
12:     append(SW_i, T_SW);
13:     for each row r in SW_i.indexarr do
14:       W_i.V ← adjust_transfunc<agg>(W_i.V, r);
15:     end for
16:     return W_i.V;
17:   end for
18: end for
```

3.2 Incremental Sampling Algorithm

The evaluation cost of the window functions is proportional to the number of tuples in the table and in each window. Thus, the execution time increases sharply as the data size grows. Since the table size N is a constant, we propose to reduce the number of tuples checked in each window, S, by sampling.

Definition 2 (*Sampling window SW*) The sampling window contains a subset of rows in window W_i and is defined by a triple $SW_i(sh, st, indexarr)$, where i represents the i-th sampling window, *indexarr* is an array which contains all the indexes of sampled tuples, *sh* is the head of sampling window, i.e., the first sampled tuple's position in *indexarr* and *st* is the tail of sampling window.

Algorithm 2 shows the procedure of incremental sampling. Firstly, we reuse the previous structure SW_{i-1} (Line 4). Then tuples, which are beyond the range of current window, are removed from SW_i (Line 6). Function *remove_head* is used to remove the first item in $SW_i.indexarr$ and update $SW_i.sh$ depending on how many tuples are removed. Then we use a temporary structure *T_SW* to store the indexes of newly sampled tuples in the range from $SW_i.indexarr[SW_i.st]$ to $W_i.t$ (Line 9–11). Next, we append the indexes in $T\_SW.indexarr$ to SW_i and update $SW_i.st$ (Line 12). Finally, we aggregate the tuples in SW_i to estimate the final result (Line 13–14).

4 Global Sampling Algorithms

All the algorithms we have described in above sections maintain a complete result set, which means we have to reorder all the tuples and calculate results for each one. When facing large data processing problems, these algorithms seem powerless. However, complete results are not always needed in statistics. On the one hand, there is a similarity between adjacent data results, due to the large number of overlaps between adjacent windows. For example, as shown in Fig. 3, W_2 and W_3 are adjacent windows and there is only one data that is different. If the two windows are big enough, the gap between them is minimal. On the other hand, the user only needs to obtain a general description of the whole and pays no attention to details in most statistical requirements scenes. For example, if you want to know the distribution of price of all the orders in an online sales website, you don't need to know the details of each order and a sample can reflect the overall situation.

Hence, this provides us with a wider range of optimization. We can generate a subset of original data and all operations are based on the subsets, which reduces the cost of sort and calculation. According to this idea, we propose two optimization algorithms for ROW mode and RANGE mode: row-labeled sampling algorithm (RLSA) and range-based global sampling algorithm (RGSA).

The reason why we build different algorithms for ROW and RANGE is that the adaptability of them is different in sampling environment. As is shown in Fig. 4, suppose the current tuple is 7 (i.e., the red number), there are two windows which are defined in the same form (i.e., BETWEEN 2 PRECEDING AND 2 FOLLOWING), but they are in different modes. The green bracket represents the window in ROW mode, while the blue one represents the window in RANGE mode. Therefore, when the windows are used on original data, there are 5 tuples in ROW window (i.e., 6,6.5,7,7.8,8) and 7 tuples in RANGE window (i.e., 5.6,6,6.5,7,7.8,8,8.5). However, when we apply the same windows on the subset which is generated from original data by using random sampling, there are still 5 tuples in ROW window (i.e., 4,6,7,7.8,8.5) and only 4 tuples in RANGE window (i.e., 6,7,7.8,8.5). And we find ROW window on subset contains 2 dirty tuples that don't belong to original ROW window (i.e., the yellow tuples 4 and 8.5), but it doesn't happen in RANGE mode.

4.1 Range-Based Global Sampling Algorithm

The first improved algorithm is designed for window in RANGE mode, named range-based global sampling algorithm (RGSA). As we discuss above, range mode is able to ensure that the data is clean, which is friendly in statistical calculations. And we don't need to adjust the window boundaries, because the nature of the range window ensures that the exact boundary can be found. So all operations are the same as dealing with normal windows. For more information, refer to algorithm 3. Here we just give the basic version.

In algorithm 3, we firstly generate a sub-dataset *ST* by using the specified sample rate (Line 1–2). Then the sub-dataset is processed in the same way as algorithm 1: partition, order, calculate. However, we have to use adjust_transfunc to replace the normal one (Line 9 and Line 13). And adjust_transfunc accepts three parameters (i.e., current value, sampling rate and current tuple) to adjust the temporary window value correctly.

Algorithm 3 Range-based Global Sampling Algorithm

Input: the whole table T and the sampling rate s
Output: approximate result for each tuple in the sampling table

```
1:  generate a new table ST by random sampling;
2:  partition and order for ST
3:  for each partition in the ST do
4:      for each window W_i in the partition do
5:          init W_i.h,W_i.t,W_i.V;
6:          if W_i.h=W_{i-1}.h then
7:              W_i.V ← W_{i-1}.V;
8:              for each row r in (W_{i-1}.t, W_i.t] do
9:                  W_i.V ← adjust_transfunc(W_i.V, s, r);
10:             end for
11:         else
12:             for each row r in [W_i.h, W_i.t] do
13:                 W_i.V ← adjust_transfunc(W_i.V, s, r);
14:             end for
15:         end if
16:         return W_i.V;
17:     end for
18: end for
```

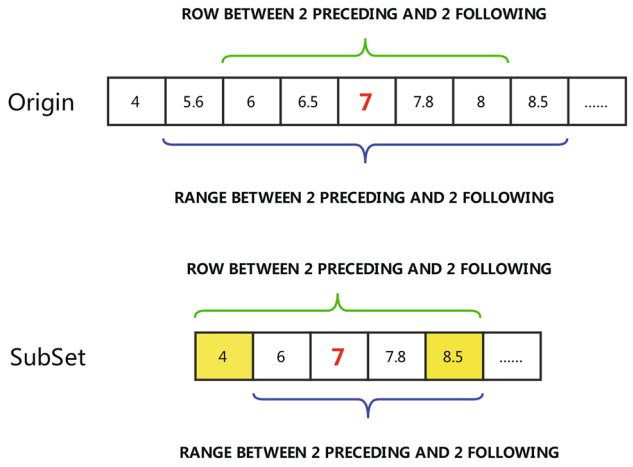

Fig. 4 Differences between ROW and RANGE

method can work, but the fatal flaw is that the dirty data can't be completely avoided. As shown in Fig. 5, although we have changed the size of window from 2 to 1, there exists a dirty tuple (tuple 8.5).

In order to eliminate dirty data completely, we design row-labeled sampling algorithm (RLSA). This algorithm doesn't sample from the origin data directly, but sample from the sorted data after first phase, when each tuple is in the final correct position. Unlike the RGSA algorithm, a new column of attributes is added to each sampling data to record the subscript position. Then the boundaries of the window can be determined accurately according to the subscript value. In other words, a window function in row mode is converted to range mode based on subscript.

Algorithm 4 Row-labeled Sampling Algorithm

Input: the whole table T and the sampling rate s
Output: approximate result for each tuple in the sampling table

1: partition and order for T
2: generate a new table ST by random sampling;
3: **for** each *partition* in the ST **do**
4: **for** each row r_i in the *partition* **do**
5: init W_i based on r_i and confirm h and t of W_i;
6: $current\_position \leftarrow$ getPosition(r_i)
7: **while** the subscript of $r_{current\_position} > W_i.h$ **do**
8: $W_i.V \leftarrow$ adjust_transfunc($W_i.V$, s, r);
9: $current\_position \leftarrow current\_position - 1$;
10: **end while**
11: $current\_position \leftarrow$ getPosition(r_i) + 1
12: **while** the subscript of $r_{current\_position} < W_i.t$ **do**
13: $W_i.V \leftarrow$ adjust_transfunc($W_i.V$, s, r);
14: $current\_position \leftarrow current\_position + 1$;
15: **end while**
16: **return** $W_i.V$;
17: **end for**
18: **end for**

Besides, because RGSA has the same characteristics as native algorithm, we can use some improved methods to optimize RGSA. Almost all the optimized algorithms in work [10, 11, 20] can be used to RGSA with simple change.

4.2 Row-Labeled Sampling Algorithm

Another improved algorithm is designed for window in ROW mode. As we discuss above, there exist dirty tuples in ROW window. And these tuples influence the accuracy of estimated results. Hence, in order to reduce or eliminate adverse effects, an intuitive idea is that we can dynamically adjust the window boundaries according to the sampling rate. For example, suppose the sampling rate is 0.5, we can reduce the window size by half. Sometimes this simple

Algorithm 4 shows the details of RLSA. Firstly, the data table T is partitioned and sorted according to the normal process (Line 1). After that, the position of each data in T will not be changed and we generate a sub-table ST by random sampling with rate s (Line 2). It should be explained that not only the source data are sampled, but also the index of the sampled data in table S is recorded. Then we calculate the boundaries of each window based on each row in ST and get the subscript of r (Line 3-6). Next, traverse data from the current row to the both sides until the subscript exceeds window range and invoke *adjust_transfunc* for each row that is traversed (Line 7–15). In fact, it's a range style computing process. Finally, return the final results (Line 16).

Figure 6 shows the new diagram of previous example by using RLSA. To make it easy to understand, we mark the

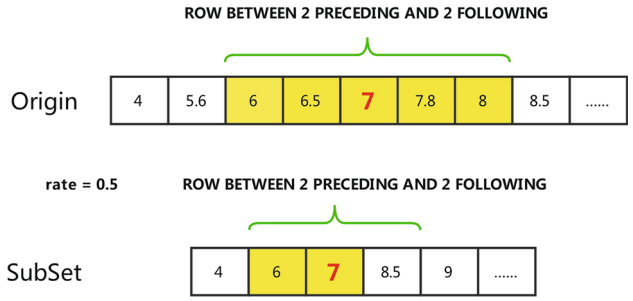

Fig. 5 An instance of dirty data

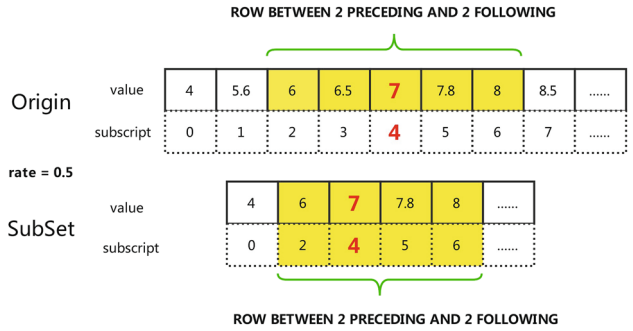

Fig. 6 An example of RLSA

subscript of each data in dashed frame. Then we can find that a new subscript attribute is added to the subset(i.e., the sample set) to identify the original position of each tuple. In this example, the current window is based on tuple 7 whose subscript is 4 and the size of window is 2, so the subscript of low bound and high bound is 2 and 6. Next we just need to find the tuples whose subscript satisfy boundary constrains (i.e., greater than or equal to 2, less than or equal to 6). Finally, 4 tuples (i.e., 6,7,7.8,8) are found in the subset and all of them belong to the original window. Hence, the dirty data can be completely avoided by this method.

4.3 Estimator and Confidence Interval

Both RGSA and RLSA sample randomly from the original space, and we can use statistical analysis method to calculate errors. Confidence interval is an important component of approximation algorithms and indicates the proximity of each approximate result to extra result. Here we give the formulae related to the estimator and confidence interval of *SUM*.

Consider a window W containing m tuples, denoted by $t_1, t_2, \ldots, t_m$. Suppose $v(i)$ is an auxiliary function whose result is equal to m times the value of t_i when applied to t_i. Then we define a new function $F(f)$, and the extra result α of sum function is equal to $F(v)$.

$$F(f) = \frac{1}{m}\sum_{i=1}^{m} f(i) \tag{1}$$

Now we sample tuples from window W and get a sequence of tuples, denoted as $T_1, T_2, \ldots, T_n$, after n tuples are selected. Then estimated result $\overline{Y_n}$ can be calculated by using Eq. (2). $\overline{Y_n}$ is the average value of the samples and is equal to $T_n(v)$.

$$T_n(f) = \frac{1}{n}\sum_{i=1}^{n} f(T_i) \tag{2}$$

The formula of sample variance is shown as Eq. (3). $T_n(f)$ is the average value of the samples. So the sample variance $\widetilde{\sigma^2}$ is equal to $T_{n,2}(v)$.

$$T_{n,q}(f) = \frac{1}{n-1}\sum_{i=1}^{n}(f(T_i) - T_n(f))^q \tag{3}$$

According to central limit theorem, we can conclude that the left-hand side follows normal distribution with mean 0 and variance 1.

$$\frac{\sqrt{n}(\overline{Y_n} - \alpha)}{\sqrt{T_{n,2}(v)}} \Rightarrow N(0,1) \tag{4}$$

After we get $\overline{Y_n}$ and $\widetilde{\sigma^2}$, we have

$$P\left\{\left|\overline{Y_n} - \alpha\right| \le \varepsilon\right\} \approx 2\Psi\left(\frac{\varepsilon\sqrt{n}}{\sqrt{T_{n,2}(v)}}\right) - 1 \tag{5}$$

where Ψ is the cumulative distribution function of the $\mathcal{N}(0,1)$ random variable. Finally, we define z_p as the $\frac{p+1}{2}$ quantile of the normal distribution function and set Eq. (5) equal to p. Then a confidence interval is computed as

$$\varepsilon_n = \frac{z_p\widetilde{\sigma}}{\sqrt{n}} \tag{6}$$

Other aggregation functions, such as COUNT, AVG, VAR, have the similar proof process. And [13, 15] give more details about the proofs of a few common aggregation functions. In addition, Haas and Hellerstein [28] proved that we could estimate the aggregate by using a non-independent and uniform sample method.

5 Experiments

In this section, we experimentally evaluate our implementations. We attempted to compare our implementations with commercial and open-source database systems, but unfortunately, most systems do not support sampling. Hence, we compared them with PostgreSQL's native window function implementation over the TPC-H

benchmark and explore the impact of sampling rate on the algorithms.

5.1 Experimental Setup

We run PostgreSQL on a computer of Lenovo with an Intel (R) Core (TM) i5-4460M CPU at 3.20 Hz, whose system has 24 GB 1600 MHz DDR3 of RAM. Then we implemented IRS and NRS on it. Since PostgreSQL does not fully support window functions in RANGE mode, we achieve RGSA with C++ and use gcc 5.3 as a compiler. Besides, RLSA can be regarded as special range calculation, so we also implemented it with C++.

5.1.1 Dataset

We use the TPC-H DBGEN instruction to generate two "order" tables, with 1.5 million rows and 15 million rows, respectively. There are 9 attribute columns in table "order," and it covers a wide variety of data types, such as INTEGER, DOUBLE, VARCHAR, DATE. You can access to TPC's official website for more information.

5.1.2 Comparison Algorithms

- **PG:** The default implementation of the PostgreSQL itself.
- **NRS:** The naive random sampling algorithm shown in [1].
- **IRS:** The incremental random sampling algorithm shown in algorithm 2.
- **RGSA:** The range-based global sampling algorithm shown in algorithm 3.
- **RLSA:** The row-labeled sampling algorithm shown in algorithm 4.

5.2 Experimental Results

In this section, we use the following query template in our experiments to demonstrate our implementation:

```
SELECT o_orderkey, aggf(o_totalprice) OVER
(ORDER BY col Rows/Range BETWEEN pre PRECEDING AND fel FOLLOWING)
FROM orders;
```

where aggf are aggregation functions, such as sum, avg, count, frame model is *ROW*, and the bounds of window are limited by *pre* and *fel*.

5.2.1 Comparison with PostgreSQL

In this part we compare the time efficiency of each algorithm. Figure 7 shows the average execution time of different algorithms. Figure 7a shows the result of default algorithm in PostgreSQL, NRS and IRS. The sampling rates of NRS and IRS are set to 0.3. As expected, our algorithms are far better than the default algorithm in PostgreSQL, especially dealing with large windows. In other words, NRS and IRS are more insensitive to window size. Besides, IRS is slightly better than NRS in terms of efficiency, because IRS avoids repeated sampling when calculating the window functions.

Figure 7b shows the results of RLSA with different sampling rate (i.e., red line is 0.5 and blue line is 0.3). Obviously, the execution speed of RLSA is much higher than that of default algorithm and the effect is obvious when the sampling rate is low, because lower sampling rate means smaller window size and less tuples needed to be calculated. The similar conclusion can be drawn from Fig. 7c. What's more, Fig. 7d shows the sample time of RLSA. In addition to finding that the time increases with the increase of sampling rate, we also find that the sampling time is very short and has little effect on the overall execution time.

5.2.2 Effect of Sampling Rate

In this part we compare the error levels and the influence of the parameters of each algorithm. Sampling rate is a very important parameter in our methods. Since naive random sampling (NRS) and incremental random sampling (IRS) are similar in this respect, all the pictures in Fig. 8 show the results of IRS algorithm based on the row mode window.

Figure 8 shows the results of IRS on the table with 1.5M tuples. The results show that execution time is negatively correlated with sampling rate, and error is positively correlated with it in any window size condition. What's more, Fig. 8a also shows that the slopes of the three curves are not similar and the black curve is the smoothest. It means the growth trend of time consumption with various rates is different as the size of window increases. In more details, IRS with low sampling rate is more insensitive to window size than that with high rate. This is the result of sample data management and the influence becomes larger as the sampling tuples become more.

Then from Fig. 8b, we can find that with fixed sample rate, the error becomes smaller when window size becomes bigger. It can be explained by the law of large numbers. The larger the volume of sample data is, the closer we are to the real value; the smaller the volume of sample data is, the higher the randomness is. And Fig. 8b also shows that

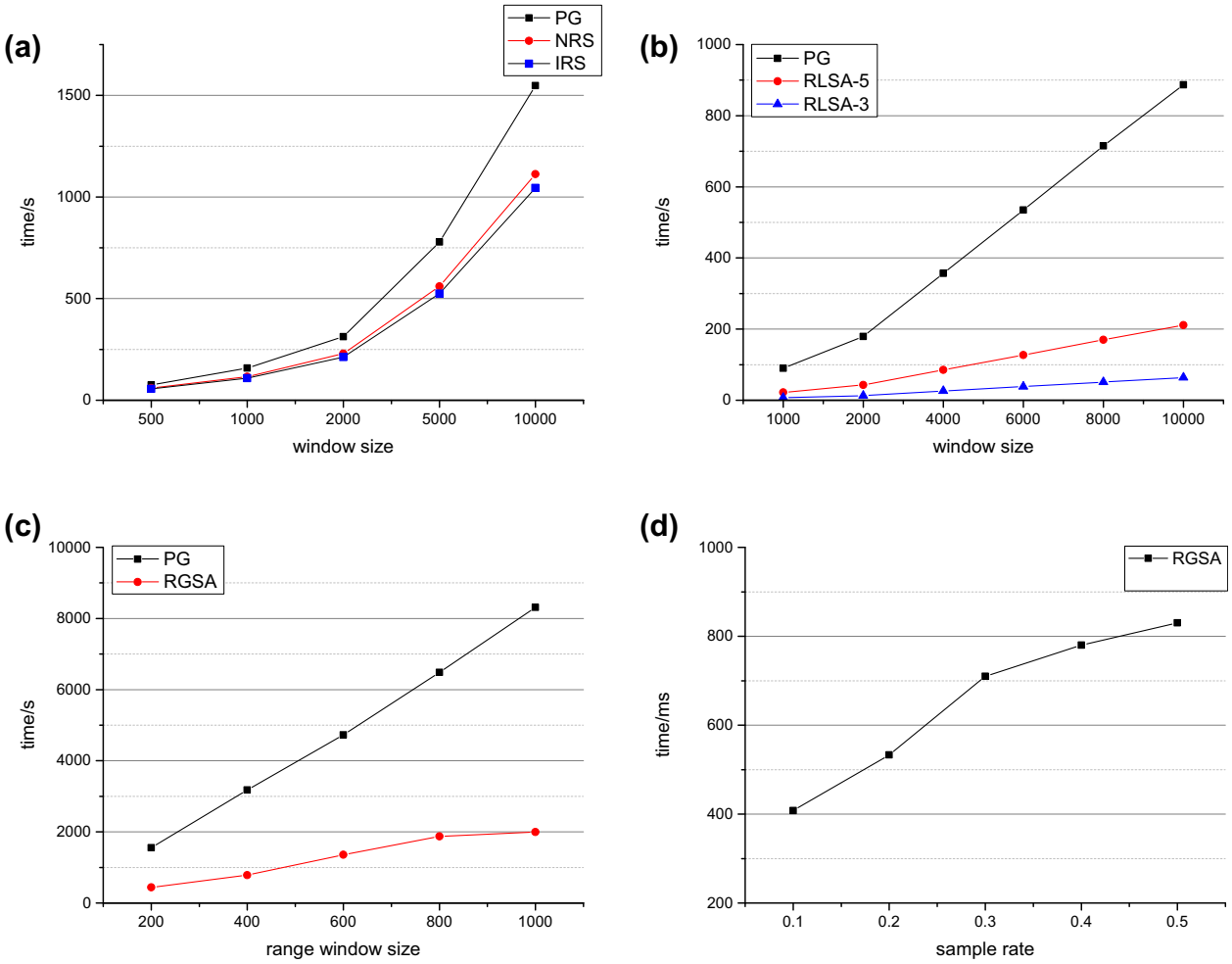

Fig. 7 Execution time of different algorithms. **a** IRS and NRS on the table with 1.5M tuples. **b** RLSA on the table with 15M tuples. **c** RGSA on the table with 15M tuples. **d** Sample time of RLSA on the table with 15M tuples

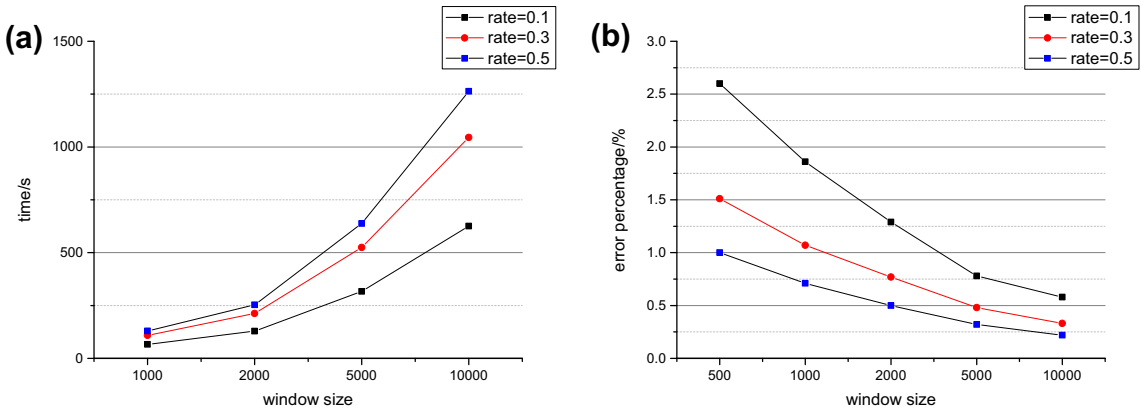

Fig. 8 Execution time and error of IRS. **a** The execution time of different sampling rates on the 1.5M tuples. **b** The error of different sampling rates on the 1.5M tuples table

the error difference among various sampling rates becomes smaller as the window becomes larger.

Next, Fig. 9a shows the results of RLSA with different sampling rate on 15M tuples table. And there are two obvious conclusions: the error decreases as the sampling rate increases; the bigger the window is, the smaller the difference in error is, which is consistent with ISR. Hence we can guide users to select lower sampling rates when

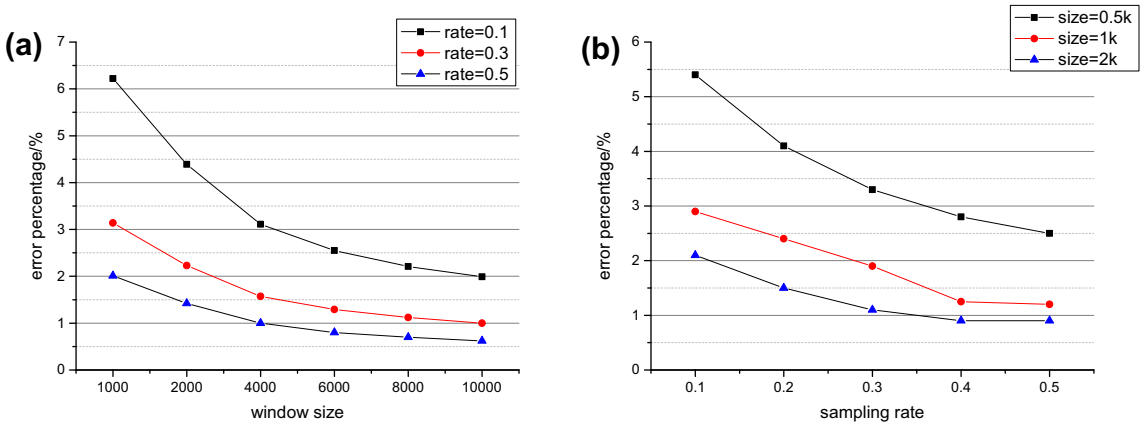

Fig. 9 Error of RLSA and RGSA. **a** The error of RLSA on the 15M tuples table. **b** The error of RGRS on the 15M tuples table

using large window size. Besides, we can find that the variation of the error is relatively stable when the sampling rate is high. The reason is that there are more tuples in each window to be sampled, which makes the estimation results closer to the true values.

What's more, Fig. 9b shows the influences of sampling rate and window size on RGRS. And we can find error of big window is more stable and is more insensitive to sampling rate than that of small window. We can also use the law of large numbers to explain these conclusions. In addition, the difference of error between the different windows is small, when the sampling rate is large.

6 Conclusion and Future Work

In this paper, we draw on some of the algorithms used in online aggregation to deal with problems of window functions. And we propose two new algorithms: range-based global sampling algorithm (RGSA) and row-labeled sampling algorithm(RLSA) and implement it in the latest version PostgreSQL. Despite the good experimental results, there are a lot of potentials that need to be exploited in the future:

- Not only RLSA and RGSA reduce the computational consumption in each window, but decrease the number of windows over the whole table. However, PostgreSQL does not fully support window functions in RANGE mode and we just implemented them with C++. So it's very useful if we could achieve RLSA and RGSA in PostgreSQL.
- The system we have implemented is not an online system and it needs users to give the sampling rate. In the future, we want our system to be more intelligent and become a real online system.

Acknowledgements This work was supported by NSFC Grants (Nos. 61532021 and 61472141), Shanghai Knowledge Service Platform Project (No. ZF1213), Shanghai Leading Academic Discipline Project (Project No. B412) and Shanghai Agriculture Applied Technology Development Program (Grant No. G20160201).

References

1. Song G, Qu W, Wang X, et al (2017) Optimizing window aggregate functions via random sampling. In: Asia-Pacific web and web-age information management joint conference on web and big data. Springer, pp 229–244
2. Zuzarte C, Pirahesh H, Ma W, Cheng Q, Liu L, Wong K (2003) Winmagic: subquery elimination using window aggregation. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, SIGMOD '03. ACM, pp 652–656
3. Bellamkonda S, Ahmed R, Witkowski A, Amor A, Zait M, Lin C-C (2009) Enhanced subquery optimizations in oracle. Proc VLDB Endow 2(2):1366–1377
4. Ben-Gan I (2012) Microsoft SQL server 2012 high-performance T-SQL using window functions. Microsoft Press, Redmond
5. Window functions for postgresql desgin overview (2008). http://www.umitanuki.net/pgsql/wfv08/design.html
6. Bellamkonda S, Bozkaya T, Ghosh B, Gupta A, Haydu J, Subramanian S, Witkowski A (2000) Analytic functions in oracle 8i. Technical report
7. Cao Y, Bramandia R, Chan CY, Tan K-L (2010) Optimized query evaluation using cooperative sorts. In: Proceedings of the 26th international conference on data engineering, ICDE, Long Beach. IEEE, pp 601–612
8. Cao Y, Chan C-Y, Li J, Tan K-L (2012) Optimization of analytic window functions. Proc VLDB Endow 5(11):1244–1255
9. Cao Y, Bramandia R, Chan C-Y, Tan K-L (2012) Sort-sharing-aware query processing. VLDB J 21(3):411–436
10. Leis V, Kan K, Kemper A et al (2015) Efficient processing of window functions in analytical SQL queries. Proc Vldb Endow 8(10):1058–1069

11. Wesley R, Xu F (2016) Incremental computation of common windowed holistic aggregates. Proc Vldb Endow 9(12):1221–1232
12. Hellerstein JM, Haas PJ, Wang HJ (1997) Online aggregation. Acm Sigmod Rec 26(2):171–182
13. Li F, Wu B, Yi K et al (2016) Wander join: online aggregation via random walks. In: Proceedings of 35th ACM SIGMOD international conference on management of data, SIGMOS'16. ACM, pp 615–629
14. Xu F, Jermaine CM, Dobra A (2015) Confidence bounds for sampling-based group by estimates. ACM Trans Database Syst 33(3):16
15. Haas PJ (1997) Large-sample and deterministic confidence intervals for online aggregation. In: Proceedings of International Conference on Scientific and Statistical Database Management. IEEE, pp 51–63
16. Wu S, Ooi BC, Tan K (2010) Continuous sampling for online aggregation over multiple queries. In: SIGMOD. ACM, pp 651–662
17. Neumann T, Moerkotte G (2004) A combined framework for grouping and order optimization. In: Proceedings of the 30st international conference on very large data bases, VLDB'04. VLDB Endowment, pp 960–971
18. Simmen D, Shekita E, Malkemus T (1996) Fundamental techniques for order optimization. In: Proceedings of the 1996 ACM SIGMOD international conference on Management of data, sigmod'96. ACM, pp. 57–67
19. Wang XY, Chernicak M (2003) Avoiding sorting and grouping in processing queries. In: Proceedings of the 29th international conference on very large data bases, VLDB'03. VLDB Endowment, pp 826–837
20. Song G, Ma J, Wang X, et al. (2017) Optimizing window aggregate functions in relational database systems. In: International conference on database systems for advanced applications. Springer, pp 343–360
21. Olken F (1993) Random sampling from databases. Ph.D. thesis, University of California at Berkeley
22. Wang L, Christensen R, Li F et al (2016) Spatial online sampling and aggregation. Proc Vldb Endow 9(3):84–95
23. Vitter JS (1985) Random sampling with a reservoir. ACM Trans Math Softw 11(1):37–57
24. Murgai SR (2006) Reference use statistics: Statistical sampling method works (University of Tennessee at Chattanooga). Southeastern Librarian, p 54
25. Adcock B, Hansen AC (2016) Generalized sampling and infinite-dimensional compressed sensing. Found Comput Math 16(5):1263–1323
26. Bengio S, Vinyals O, Jaitly N et al (2015) Scheduled sampling for sequence prediction with recurrent neural networks. In: NIPS 15 Proceedings of the 28th International Conference on Neural Information Processing Systems. MIT Press, Cambridge, pp 1171–1179
27. Qin C, Rusu F (2014) PF-OLA: a high-performance framework for parallel online aggregation. Distrib Parallel Databases 32(3):337–375
28. Haas PJ, Hellerstein JM (1999) Ripple joins fro online aggregation. In: Proceedings of the ACM SIGMOD international conference on management of data. ACM, pp 287–298
29. Jin R, Glimcher L, Jermaine C et al (2016) New sampling-based estimators for OLAP queries. In: International conference on data engineering. IEEE, pp 18
30. Joshi S, Jermaine CM (2009) Sampling-based estimators for subset-based queries. VLDB J 18(1):181–202
31. Wang B, Yang X, Wang G et al (2016) Energy efficient approximate self-adaptive data collection in wireless sensor networks. Front Comput Sci 10(5):936–950
32. Zhu R, Wang B, Yang X et al (2017) SAP: improving continuous top-K queries over streaming data. IEEE Trans Knowl Data Eng 29(6):1310–1328
33. Fan Q, Tan KL (2015) Towards window analytics over large-scale graphs. In: Proceedings of the 2015 ACM SIGMOD on Ph.D. symposium. ACM, pp 15–19