 CMC, vol.60, no.1, pp.117-132, 2019

A Scalable Method of Maintaining Order Statistics for Big Data Stream

Zhaohui Zhang*[,1,2,3] , Jian Chen[1], Ligong Chen[1], Qiuwen Liu[1], Lijun Yang[1], Pengwei Wang[1,2,3] and Yongjun Zheng[4]

Abstract: Recently, there are some online quantile algorithms that work on how to analyze the order statistics about the high-volume and high-velocity data stream, but the drawback of these algorithms is not scalable because they take the GK algorithm as the subroutine, which is not known to be mergeable. Another drawback is that they can't maintain the correctness, which means the error will increase during the process of the window sliding. In this paper, we use a novel data structure to store the sketch that maintains the order statistics over sliding windows. Therefore three algorithms have been proposed based on the data structure. And the fixed-size window algorithm can keep the sketch of the last W elements. It is also scalable because of the mergeable property. The time-based window algorithm can always keep the sketch of the data in the last T time units. Finally, we provide the window aggregation algorithm which can help extend our algorithm into the distributed system. This provides a speed performance boost and makes it more suitable for modern applications such as system/network monitoring and anomaly detection. The experimental results show that our algorithm can not only achieve acceptable performance but also can actually maintain the correctness and be mergeable.

Keywords: Big data stream, online analytical processing, sliding windows, mergeable data sketches.

1 Introduction

The traditional application is built on the concept of persistent data sets that are stored reliably in stable storage and queried or updated several times throughout their lifetime [Zhang, Zhang, Wang et al. (2018)]. Nowadays, large volumes of stream data arise rapidly,

[1] School of Computer Science and Technology, Donghua University, 201620, China.
[2] The Key Laboratory of Embedded System and Service Computing, Ministry of Education, Tongji University, 201804, China.
[3] Shanghai Engineering Research Center of Network Information Services, 201804, China.
[4] School of Electronics, Computing and Mathematics, University of Derby, Derby, United Kingdom.
* Corresponding Author: Zhaohui Zhang. Email: zhzhang@dhu.edu.cn.

such as transactions in bank account [Zhang, Zhou, Zhang et al. (2018)] or e-commerce business [Yu, Ding, Liu et al. (2018)], credit card operations, data collection in Internet of Things (IoT) [Miao, Liu, Xu et al. (2018)], information in disaster management systems [Wu, Yan, Liu et al. (2015); Xu, Zhang, Liu et al. (2012); Wang, Zhang and Pengwei (2018)], behavior data analysis and anomaly detection in large-scale network service system [Zhang, Ge, Wang et al. (2017); Zhang and Cui (2017)] and data mining in live streaming [Li, Zhang, Xu et al. (2018)]etc. Analysis of order statistics plays an important role in analyzing data stream, which can help us to know the distribution of the data, make decisions, detect the anomaly data or help further data mining. Within applications, the high-volume and high-velocity features and limited memory make data stream pass only once, which means we can not store all the data in the memory and access the data that is already passed away. But if an approximate answer is acceptable, there are some online quantile algorithms that can maintain order statistics over data stream in the sketch and then we can get the approximate answer by querying the sketch. Combined with the sliding window, the online quantile algorithm can help us understand the order statistics or distribution of the recent data in the data stream.

In this paper, we propose a method that can maintain the data stream order statistics over the sliding window including fixed-size window and time-based window. A sketch will be created to store the stream data over the sliding window by our algorithm. Within a certain range of errors, we can get the result of quantile query or rank query from the sketch in a very short time. Compared to other algorithms that have been used to solve this kind of problem [Lin, Lu, Xu et al. (2004); Tangwongsan, Hirzel and Schneider (2018)], the advantages of our algorithm are the correctness-the error will not increase during the window updating and mergeable property-the sketches of two windows can be merged into one sketch.

And the remainder of this paper is constructed as follows. Section II introduces the related work that have been done till now. Section III gives some definitions that will be used in this paper and the basic data structure that we will use. Section IV represents our method to solve quantiles problem on the sliding window, including the basic structure, fixed-size window algorithm, time-based algorithm, and window aggregation. In section V, we designed some experiments to test the performance of our algorithm. And the last section VI gives conclusions of our work and describes the future work.

2 Related work

For the quantile problem, there are two surveys [Wang, Luo, Yi et al. (2013); Greenwald and Khanna (2016)] explain the status of research in terms of theory and algorithm in a very easy-to-understand way. In long-term research of quantile problem, there exist several algorithms to solve this problem. Greenwald et al. [Greenwald and Khanna (2001)] created an intricate deterministic algorithm (GK) that requires $\mathcal{O}((\frac{1}{\epsilon})\log(\epsilon n))$ space. This method improved upon a deterministic (MRL) summary of Manku et al. [Manku, Rajagopalan and Lindsay (1998)] and a summary implied by Munro et al. [Munro and Paterson (1978)]

which use $\mathcal{O}((\frac{1}{\epsilon})(log\epsilon n)^2)$ space. But Agarwal, et al. [Agarwal, Cormode, Huang et al. (2013)] prove that GK algorithm is not fully mergeable. Karnin et al. [Karnin, Lang and Liberty (2016)] created the optimal quantile algorithm as KLL, the best version of this algorithm requires $\mathcal{O}((\frac{1}{\epsilon})loglog(\frac{1}{\delta})$ The KLL algorithm is considered to be the optimal quantile algorithm until now both in terms of space usage. Considering the sliding window, Lin et al. [Lin, Lu, Xu et al. (2004)] was the first one to propose the quantile approximation solution in the sliding window model. And he achieved space usage $\mathcal{O}(\frac{log(\epsilon W)}{\epsilon} + \frac{1}{\epsilon^2})$. Arasu et al. [Arasu and Manku (2004)] improved this to $\mathcal{O}(\frac{1}{\epsilon}log\frac{1}{\epsilon}logW)$. These algorithms are all based on the idea that split the window into small chunks and then use quantile algorithm to summarize each chunk. Yu et al. [Yu, Crouch, Chen et al. (2016)] proposed a sliding window algorithm called exponential histograms and used the GK algorithm as a subroutine, which is considered the best algorithm over sliding window until now, to the author's knowledge. But the approximation error will increase during the updating of the sliding window. For the window aggregation problem, a survey [Tangwongsan, Hirzel and Schneider (2018)] explain the theory and the current methods. Odysseas et al. [Papapetrou, Garofalakis and Deligiannakis (2012)] create a method can sketch distributed sliding-window data streams.

Considering the SW-GK is the optimal algorithm till now and it achieves great performance on the insert time and query time. But the drawback of this algorithm is that the window update way of SW-GK is to merge two structures and the approximation error will increase after each merge operation. The algorithms have been mentioned above take GK algorithm as a subroutine which is known don't have the mergeable property, so these algorithms can't aggregate different windows, which means they are not scalable.

3 Definition and model

3.1 Quantile and rank

Quantile and rank are both order statistics of data: the quantile ϕ(x) of a set N is an element x such that $\phi \mid N \mid$ elements of N are less than or equal to x. Given a set of elements x_1, . . . , x_n, the rank of x in a stream N as $R(x) = \phi \mid N \mid$, which represents the number of elements such that $x_i \leq x$. And the quantile of a value x is the fraction of elements in the stream such that $x_i \leq x$.

In the ϵ-approximate problem, the data stream has N numeric elements. An additive error ϵn for R(x) is an ϵ approximation of its rank. In addition, when we query for a ϕ-quantile, where $0 \leq \phi \leq 1$, we will get the result that is guaranteed to be in the $[\phi - \epsilon, \phi + \epsilon]$ quantile range.

3.2 Sliding window

A data stream is a sequence of data elements available for a period of time. At any point in time, a sliding window over a stream is a bag of last W elements of the stream seen so far. To help us understand the recent data, we consider two types of sliding windows,

the fixed-size sliding window whose window size is fixed and the time-based window whose window size varies over time. Formally, both types of windows are modeled using two basic operations-insert operation (insert a new element into the window) and delete operation(delete the oldest element from the window).

3.3 Basic structure

We firstly begin with the work of Karnin et al. [Karnin, Lang and Liberty (2016); Agarwal, Cormode, Huang et al. (2013)], which uses a special data structure to store the data over the whole stream. Here is the basic data structure of the algorithm-a compactor. A compactor can store k elements and each element has a weight of w. Different layer compactor has different capacity and weight. The compactor has a compaction operation, which can compact its k elements into $k/2$ elements of weight $2w$. When the compactor finishes the compaction operation, we feed the results into the next layer compactor and so on. To maintain the order statistics and answer the quantile query, the requirement is that the elements in the compactor need to be in the order before compaction operation. During the compaction operation, either the even or the odd elements (based on the index) in the sequence are chosen. The unchosen elements are discarded, while the weight of the chosen elements is doubled. The error of the rank estimation before and after the compaction defers by at most w regardless of k. Fig. 1 shows a simple example of a compactor, if its rank of a query in the compactor is even, the rank is unchanged. If it is odd, the rank is increased or decreased by w with equal probability. Fig. 2 shows the structure of the compactors. The new element first comes to the first layer. When the first layer is full, compact the data in this layer and put the results into the second layer. When the second layer is full, do the similar operation and so on.

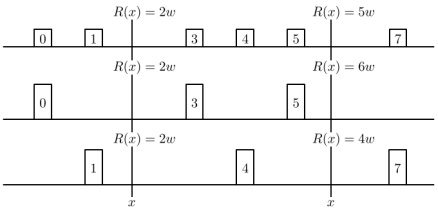

Figure 1: A example of a compactor

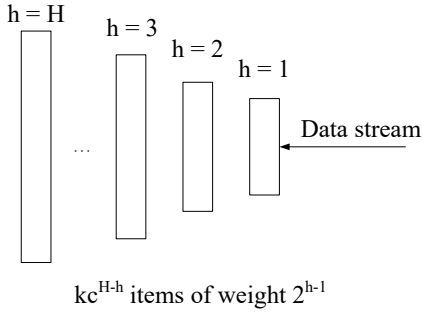

Figure 2: Compactors structure

We define the H as the numbers of layers of compactors and each compactor has its own capacity, denoted by k_h with the indexes by their height $h \in 1, ..., H$. The weight of elements at height h is $w_h = 2^{h-1}$.Considering the requirement of the compaction operation, the capacity of smallest compactor need be at least 2. For brevity, we set $k = k_H$. It gives that $k_h \geq kc^{H-h}$ for $c \in (0.5, 1)$.

Lemma 3.1. There exists an algorithm that can compute an ϵ approximate for the rank or quantile problem whose space complexity is $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)}+log(\epsilon n))$. This algorithm also produces mergeable summaries [Karnin, Lang and Liberty (2016)].

4 Method

4.1 Fixed-size window algorithm

For the fixed-size window, the insert operation is always combined with the delete operation. But we use the compactors to store the data, the element in different compactor has different weight with respect to the height, which means that one element doesn't represent itself, it has the weight, it records the number of compaction operation. So in the fixed-size window situation, we could not just delete the oldest element when the new element comes. Then we try to use a special array of size H to help to determine if it is the time to discard the oldest element in the sliding window. The index of this array is based on the height of the compactors. When the array of one layer count reaches to 2, turn on the trigger of this layer. When the trigger of the highest layer is on, it means it can discard the oldest element in the sliding window.

Here is the algorithm that we combine the insert operation and delete operation together. First, we have two conditions, one is that the current window is not full, which means we can continue to put the data into the window. Another is that the current window is full, which means the window sketch has represented W elements, with the new element comes, the oldest element should be discarded from the sketch. Note that the oldest element must be in the highest compactor, for example, k_H represents the capacity of the highest compactor and w_H represents the weight of the per element in the highest compactor. Our goal is to discard the oldest element in the window, according to the structure, when the new $2w_H$ elements come in, theoretically, the oldest element in the sketch can be discarded. According to the compaction operation, for every layer, the data are in the order, we can compact the oldest element in this layer with its surrounding element(left or right with the same probability), so we use a H size array to trigger compaction operation. For the highest compactor, we pretend to do one compaction operation for the two elements-just deleting the oldest two elements. For other compactors, if the trigger is on, find the oldest element and its neighbor, discard one with the same probability and insert the other into the next compactor. In the end, update the trigger array. Note that the insert operation is finding the correct position to keep elements in the order in the compactors including insert operation on into the first layer. Fig. 3 and Algorithm 1 describes the process of updating fixed-size window. For each layer, when the trigger is on, find the oldest element and its neighbor, discard one of them with equal probability and put the other into the next layer. For the last layer, after $2w_H$ elements come, the trigger of this layer is on, discard the oldest two elements. The red circle represents the discarded one and yellow circle represents the one should be put into the next layer.

Theorem 4.1. Our algorithm with W size window, has space complexity $\mathcal{O}((\frac{1}{\epsilon})\sqrt{log(\frac{1}{\epsilon})}+$

Algorithm 1 fixed-size window algorithm

```
1: procedure ADD(item)                                              ▷
2:     count ← count + 1
3:     if count < W then Sketch.update(item)
4:     else if count = W then
5:         for h = 0 → H do SortByValue(compactors[h])
6:     else
7:         trigger[0] ++
8:         for h = 0 → H do
9:             if h = H then
10:                if trigger[h] = 2 then
11:                    DeleteTwoOledst(h)
12:            else
13:                if trigger[h] = 2 then
14:                    One ← FindOldest(h)
15:                    Two ← NearBy(One)
16:                    if Random < 0.5 then
17:                        Choose ← One
18:                    else
19:                        Choose ← Two
20:                    Delete(One, Two)
21:                    Insert(h + 1, Choose)
22:                    trigger[h + 1] + +
23:                    trigger[h] ← 0
```

$log(\epsilon W))$. And the update time complexity is $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)})$ and query complexity is $\mathcal{O}(log(\frac{1}{\epsilon}\sqrt{log(\frac{1}{\epsilon})})(log(W) - log(\frac{1}{\epsilon}\sqrt{log(\frac{1}{\epsilon})})$. The correctness of our algorithm is unchanged, which can keep the rank query procedure still returns a value v with rank between $(q - \epsilon)W$ and $(q + \epsilon)W$.

Proof. When the window is full, it means that this data structure is stable and this sketch represent W elements. Then we look at the data structure.

Firstly, we know that the second compactor from the top compacted its elements at least once. Therefore $W/k_{H-1}w_{H-1} \geq 1$ which gives

$$H \leq log(W/k_{H-1}) + 2 \leq log(W/ck) + 2 \tag{1}$$

k_h represents the capacity of the compactor at height h and $w_h = 2^{h-1}$ represents the weight of the per element in this compactor. Then we define m_h to represent the number

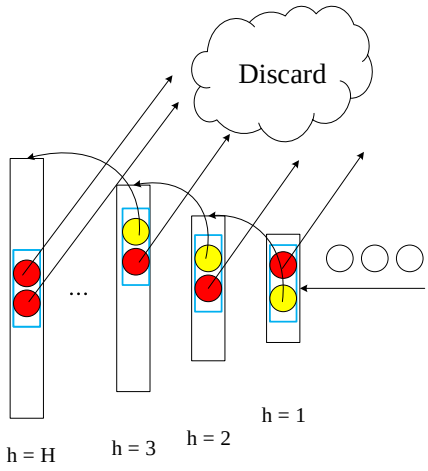

Figure 3: The process of updating fixed-size window

of compaction operations at height h.

$$m_h \leq \frac{W}{k_h w_h} \leq \frac{2W}{k2^H}(\frac{2}{c})^{H-h} \leq (\frac{2}{c})^{H-h-1} \tag{2}$$

Then we use $R(k,h)$ to represent the rank of x at height h. Note that each compaction operation in layer h either leaves the rank of x unchanged or adds w_h or subtract w_h with equal probability. Therefore, $err(x,h) = R(x,h) - R(x,h-1) = \sum_{i=1}^{m_h} w_h X_{i,h}$, where $\mathbb{E}[X_{i,h}] = 0$ and $|X_{i,h}| \leq 1$. The final discrepancy between real rank of x and its our approximate rank $\widetilde{R}(x) = R(x,H)$ is

$$R(x,H) - R(x,0) = \sum_{h=1}^{H} R(x,h) - R(x,h-1) = \sum_{h=1}^{H}\sum_{i=1}^{m_h} w_h X_{i,h}. \tag{3}$$

Lemma 4.1 (Hoeffding)**.** Let $x_1, ..., X_m$ be independent random variables, each with an expected value of zero, taking values in the range $[-w_i, w_i]$. Then for any t > 0, we got

$$Pr[|\sum_{i=1}^{m} X_i| > t] \leq 2exp(-\frac{t^2}{2\sum_{i=1}^{m} {w_i}^2}) \tag{4}$$

According to the Hoeffding's inequality, and we let ϵW be the total error. We can get the inequality below:

$$\begin{aligned} Pr[|R(x,H)-R(x,0)|\geq \epsilon W] &= Pr[\sum_{h=1}^{H}\sum_{i=1}^{m_h} w_h X_{i,h} > \epsilon W] \\ &\leq 2exp(-\frac{\epsilon^2 W^2}{2\sum_{h=1}^{H}\sum_{i=1}^{m_h} {w_h}^2}) \end{aligned} \tag{5}$$

A computation shows that

$$\sum_{h=1}^{H}\sum_{i=1}^{m_h} {w_h}^2 = \sum_{h=1}^{H} m_h {w_h}^2 \leq \sum_{h=1}^{H} (\frac{2}{c})^{H-h-1} 2^{2h-1} \leq \frac{(2/c)^{H-1}}{4}\frac{(2c)^H}{(2c-1)} \leq \frac{c}{8(2c-1)}2^{2H} \tag{6}$$

Substituting Eqs. (1) and (6) into Eq. (5) and setting $C=(2c-1)c/4$ we get the inequality

$$Pr[|R(x,H)-R(x,0)|\geq \epsilon W] \leq 2exp(-C\epsilon^2 k^2) \tag{7}$$

Note that the algorithm has H layers and $k_h \geq kc^{H-h}$, $c \in (0.5,1)$, we let $k_h = \lceil kc^{H-h} \rceil + 1$. In our algorithm, the total space usage includes two parts: the compactors and the trigger array, which is $\mathcal{O}(\sum_{h=1}^{H} k_h + 2H)$

$$\sum_{h=1}^{H} k_h + 2H \leq k/(1-c) + 4H = \mathcal{O}(k + log(W/k) \tag{8}$$

According to Eq. (7) and requiring failure probability at most δ we conclude that it suffices to set $k = (C/\epsilon)\sqrt{log(2/\delta)}$. Then we set $\delta = \Omega(\epsilon)$ suffices to union bound over the failure probabilities of $\mathcal{O}(1/\epsilon)$ different quantiles. This provides a fixed-window algorithm for the quantiles problem of space $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)} + log(\epsilon W))$

When the window is full, according to the algorithm, we can find that after two elements come, we need do one find oldest operation based on the time data arrives and one insert operation which inserts the chosen data into the next layer, which needs to find the correct position to make the elements in this layer are still in the order. In the first layer, we need to traverse the elements of this layer to find the oldest element and use binary search in the second layer to find the position to insert. After four elements come, there will be one find operation and one insert operation in the second layer and two find operations and two

insert operations in the first layer, etc. What's more, we need also mention that inserting every element into the first layer also need to keep all the elements in the order in the first layer. So we can get the time cost per m elements, $m \times k_1 + \frac{m}{2}(k_1 + log(k_2)) + \frac{m}{2^2}(k_2 + log(k_3)) + \frac{m}{2^3}(k_3 + log(k_4)) + ... + \frac{m}{2^{H-1}}(k_{H-1} + log(k_H)) + \frac{m}{2^H}k_H$. Based on the knowledge $k_h \leq kc^{H-h}$ and $c \in (0.5, 1)$, the amortized time is

$$
\begin{aligned}
&m^{-1} \times \left(m \times k_1 + \textstyle\sum_{h=1}^{H-1}(k_h + log(k_{h+1}))\frac{m}{2^h} + \frac{m}{2^H}K_H\right) \\
&\leq m^{-1} \times \left(m \times k_1 + \textstyle\sum_{h=1}^{H-1}(kc^{H-h} + log(kc^{H-h-1}))\frac{m}{2^h} + \frac{m}{2^H}k\right) \\
&= kc^{H-1} + log(k)\textstyle\sum_{h=1}^{H-1}((\frac{1}{2})^h) + k\sum_{h=1}^{H}(c^H(\frac{1}{2c})^h) + \sum_{h=1}^{H-1}((\frac{1}{2})^h log(c^{H-1-h})) \\
&= kc^{H-1} + log(k)(1 - (\tfrac{1}{2})^{H-1}) + k(\tfrac{c^H(1-(\frac{1}{2c})^H)}{2c-1}) \\
&+ log(c)(H - 3 + \tfrac{1}{2^{H-2}}) = \mathcal{O}(k)
\end{aligned}
\tag{9}
$$

According to the previous knowledge about k, we can get that the final update time complexity is $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)})$.

The query operation is to find all the elements stored in the compactors which are less than the given value and sum their weights together. Considering the compactors always contains the sorted elements, we could use the binary search to find the elements less than the given one, which gives the query time

$$
\sum_{h=1}^{H}(log(k_h)) \leq \sum_{h=1}^{H}(log(ck^{H-h})) = Hlog(k) + \frac{H(H-1)log(c)}{2} = \mathcal{O}(Hlog(k)) \tag{10}
$$

Considering that H can be represent by the $\mathcal{O}(log(\frac{W}{k})log(k))$, the final query time complexity will be $\mathcal{O}(log(\frac{1}{\epsilon}\sqrt{log(\frac{1}{\epsilon})})(log(W) - log(\frac{1}{\epsilon}\sqrt{log(\frac{1}{\epsilon})})$

During the execution of our algorithm, the height of compactors is not changed. Every compactor can have two more element at most and each compactor still leaves the rank of the x unchanged or add w_h or subtract w_h with equal probability. For the layer h, the $err(x, h)$ is unchanged. So the total error is still $\sum_{h=1}^{H} err(x, h) = \sum_{h=1}^{H}\sum_{i=1}^{m_h} w_h X_{i,h}$. This means that our algorithm can maintain the correctness that the rank query procedure returns a value v with a rank between $(q - \epsilon)W$ and $(q + \epsilon)W$. ■

4.2 Time-based window algorithm

For the time-based window, when a new item comes, put it into the first compactor. Then if the compactor is full, do the compaction operation. Based on the time the new item

contains, compute the window threshold. Then do the delete operation, for every item stored in the compactors, if its time before the threshold, delete the item.

Algorithm 2 time based window algorithm

```
1: procedure ADD2(item, timelength)
2:     Sketch.update(item)
3:     timeNow ← getTimeNow()
4:     T ← timeNow − timelength
5:     for h = 0 → H do
6:         for j = 0 → len(compactors[h]) do
7:             if compactor[h][j].time < T then
8:                 Delete(compactor[h][j])
```

Theorem 4.2. Setting W' to the elements in the current time-based window, our algorithm with time-based window has update time $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)} + log(\epsilon W'))$ and query time $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)} + log(\epsilon W'))$

Proof. Note that every update operation needs traverse all the elements to delete the expired elements. The space usage is $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)} + log(\epsilon W'))$. So the update time is $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)} + log(\epsilon W'))$. Similarly, the query operation also need to traverse all the data, which gives $\mathcal{O}((1/\epsilon)\sqrt{log(1/\epsilon)} + log(\epsilon W'))$ query time. ■

4.3 Window aggregation

Aggregating two windows means that merge the sketches of the two windows together. Firstly, let the small(the number of layer is small) one grow until it has at least as many compactors as the other. Then, Append the elements in same height compactors. Each level that contains more than k_h elements need do one compaction operation.

Algorithm 3 window aggregate algorithm

```
1: procedure AGGREGATE(s1, s2)
2:     if s1.height < s2.height then
3:         s1.grow()
4:     else
5:         s2.grow()
6:     for h = 0 → H do
7:         s1.compactor[h].add(s2.compactor[h])
8:     for h = 0 → H do
9:         if len(s1.compactor[h]) > k_h then
10:            s1.compactor[h].compact
```

According to this mergeable property, we can extend our algorithm to the distributed environment, which can handle more huge data. Here is our distributed system structure,

when the data comes, it will first come to the process point and then will be allocated to the different machines. Every machine uses our algorithm to sketch the stream data separately. Then merge all the sketches together. For example, if we want to analyze the data in the last 5 minutes. The data stream will be allocated to three nodes with Round-Robin Scheduling and every node only need to update $1/3$ data. Then after every 30 seconds, we merge all the sketches into the final sketch. In this way, we can consider the final sketch maintains the order statistics of the last 5 minutes data. If we want to do some queries, we can get the approximate result from the final sketch. Fig. 2. describes this system structure. For every window, the time resolution is 5 minutes. After every 30 seconds, we aggregate all the windows and get the final sketch. Then we query the final sketch for analyzing the order statistics of the recent 5 minutes.

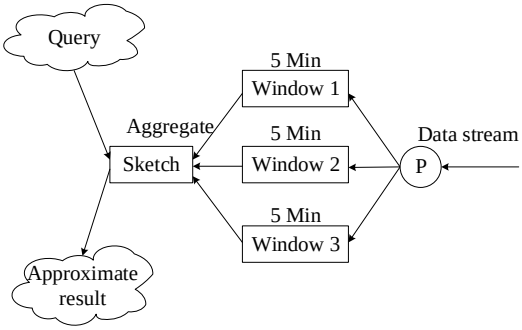

Figure 4: System structure.

5 Experiments

We mainly implemented our fixed-size algorithm by using Python and did some experiments to prove the performance of our algorithm. Experiments were run on a Core i5 2.1 GHz CPU computer with 8 Gb memory running Windows 10. The first experiment is a simple example to illustrate that our algorithm can really work. The second and third experiments are to test the insert operation and query operation time, separately. The fourth experiment is to calculate the storage that our algorithm need use. The fifth experiment is to prove the correctness of our algorithm, which means that the error between the real rank and the sketch rank will not increase during the process of the window sliding. The last experiment is to prove the mergeability of our algorithm.

5.1 A simple example

The parameters of our algorithm are k and W, where k is the maximum capacity of all compactors as we explained before and W is the window size. In this simple experiment, we set the k to 32 and W to 500. We use the Cumulative Distribution Function to describe the order statistics of the data. Fig. 5 shows that the distribution of the last 500 elements,

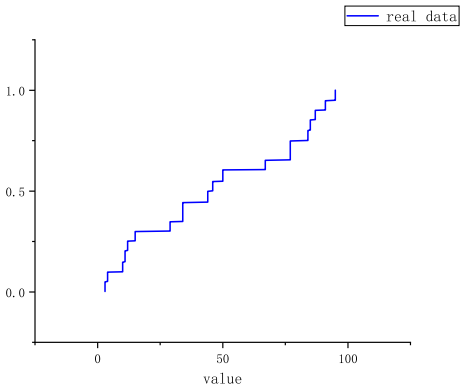

Figure 5: The CDF of the real data

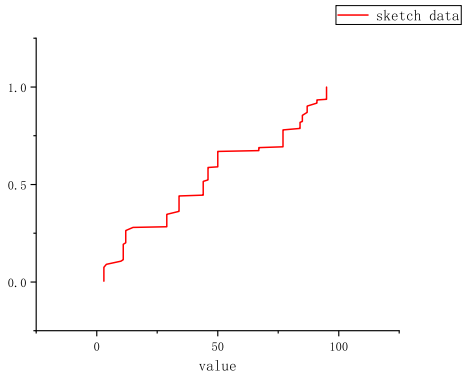

Figure 6: The CDF of the sketch data

which are the end of the data stream. And Fig. 6 describes the distribution generated by the sketch stored in our algorithm of the last 500 elements. Comparing Fig. 5 and Fig. 6, we can see that these two distributions are pretty similar, which means that our algorithm can actually maintain the order statistics over the sliding window.

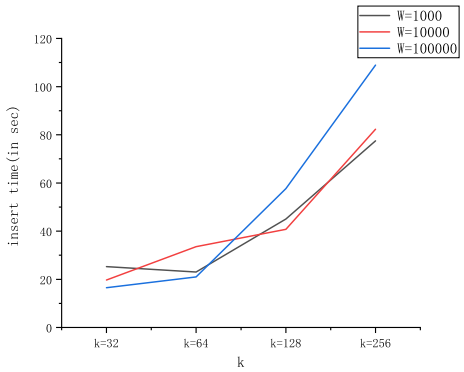

Figure 7: The insert time

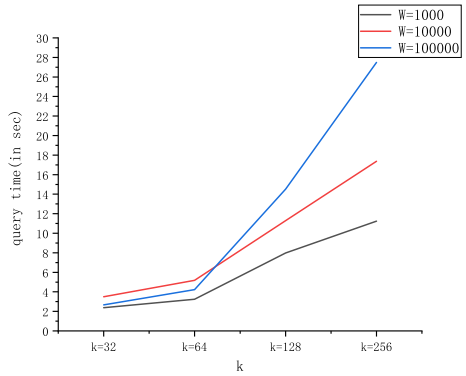

Figure 8: The query time

5.2 Insert time experiment

The previous section gives the insert time complexity by math derivation. In this section, we did an experiment to prove that the insert time is acceptable. We used a random dataset including 1000000 entries, set different parameters k and W and use this dataset to run our fixed-size sliding window algorithm. We calculate the running time as the insert time. In this experiment, we set different k, W and get different insert time. Fig. 7 shows the relationship between the insert time and k and W.

From Fig. 7, we can see that the insert time will rise with the increase of k and the window size doesn't have great influence on the insert time when the k is not big. But when k is set to 256 or much bigger, the influence of W will become longer. When the k is small, the insert time is pretty acceptable. Considering the window aggregation, we can extend

our algorithm into the distributed environment, it will reduce the insert time dramatically. It means we can always control the insert time in a range that we want.

5.3 Query time experiment

Similarly, with the insert time experiment, we use the same dataset and the same parameters. After 10 elements, we query the rank or quantile of the value of the incoming element, then calculate the total query time. Fig. 8 shows the result of the query time.

From Fig. 8, when the k is set small the query operation is very pretty quick and not easily influenced by the window size. When the k is set large, the bigger window size has longer query time.

5.4 Sketch size experiment

From another point of view, the value of k determines the accuracy of the sketch, the greater the k value, the higher the accuracy, when the k become higher, the accuracy of the insertion time, query time, storage size will increase. Fig. 9 describes the relationship between the sketch size and parameters k, W. The effect of k on storage size is greater than the value of W. For example, if we set k to 32, we may use approximately 100 elements to represent 1000 elements in the sliding window, even the window size increase to 100000 the storage size will increase to 125 – a very small influence.

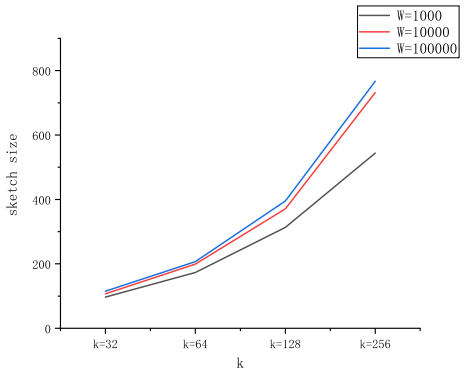

Figure 9: The sketch storage

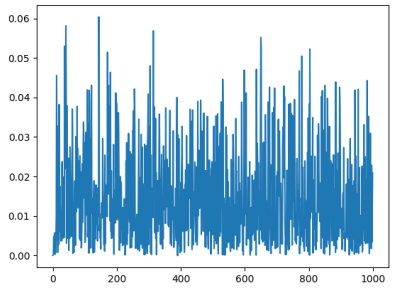

Figure 10: The correctness test

5.5 Correctness experiment

The interesting part of our algorithm is that our algorithm can maintain the same correctness – the error will not increase during the process of the window sliding. This property has been proved in the previous section. Here is a simple example to illustrate this property, we set k to 128, W to 100000 and query the rank 1000 times during the process of the window sliding. We compare the real rank and our rank from the sketch and calculate the error between them. Fig. 10 shows that the errors between real rank and our result during the

1000 queries. We can see the error is controlled in a certain range during the 1000 queries, which means the error did not increase during the process of the window sliding.

5.6 Window aggregation experiment

In the previous part, we have proposed the window aggregation algorithm, which means that the sketches of two windows can be merged into one window sketch. So in this section, we also implemented the window aggregation algorithm on our dataset. We split the stream data into two parts and each part runs one 500-size window algorithm. In the end, we merged these two sketches into one. We also use one program to run the 1000-size window algorithm. We also use CDF to describe the distribution of the data. Fig. 11 describes the result of the 1000-size window algorithm and Fig. 12 describes the result of the merged sketch of two 500-size window sketches. From Fig. 11 and Fig. 12, these two results are pretty similar and these two sketches can represent the distribution of the origin data. In this way,we also prove that our algorithm can actually be merged.

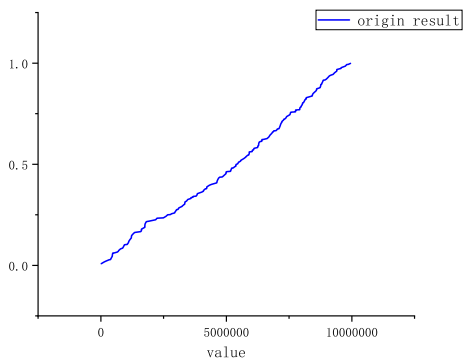

Figure 11: The CDF of 1000-size sketch

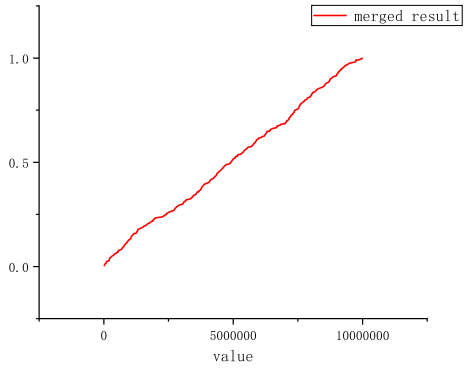

Figure 12: The CDF of merged sketch

6 Conclusions

In this paper, we propose a novel method that can maintain the stream data order statistics over the sliding fixed-size window, which can answer the quantile or rank query in a short time. The experiments show that our algorithm works properly and the insert time and query time are also both acceptable. Unlike other algorithms, our algorithm has the mergeable property and pay more attention to the correctness – the error will not increase during the process of the window sliding.

We also propose a time-based window algorithm, which is more flexible in different scenarios. In addition, the window aggregation algorithm enables parallel processing, which gives the opportunity to extend the quantile online algorithm into the distributed system. This provides a speed performance boost and makes it more suitable for modern applications such as system monitoring and anomaly detection.

For the future work, we are considering that if we accept the random method, the sampler is considered to be an effective method in the stream processing area. In the future, we are

going to combine the sampler and our algorithm together to make further improvement in our algorithm.

Acknowledgement: This work was supported by National Natural Science Foundation of China (Nos. 61472004, 61602109), Shanghai Science and Technology Innovation Action Plan Project (No. 16511100903)

References

Agarwal, P. K.; Cormode, G.; Huang, Z.; Phillips, J. M.; Wei, Z. et al. (2013): Mergeable summaries. *Transactions on Database Systems*, vol. 38, no. 4, pp. 26.

Arasu, A.; Manku, G. S. (2004): Approximate counts and quantiles over sliding windows. *Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 286-296.

Greenwald, M.; Khanna, S. (2001): Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, vol. 30, no. 2, pp. 58-66.

Greenwald, M. B.; Khanna, S. (2016): Quantiles and equi-depth histograms over streams. *Data Stream Management*, pp. 45-86.

Karnin, Z.; Lang, K.; Liberty, E. (2016): Optimal quantile approximation in streams. *Foundations of Computer Science*, pp. 71-78.

Li, D.; Zhang, G.; Xu, Z.; Lan, Y.; Shi, Y. et al. (2018): Modelling the roles of cewebrity trust and platform trust in consumers propensity of live-streaming an extended tam method. *Computers, Materials & Continua*, vol. 55, no. 1, pp. 137-137.

Lin, X.; Lu, H.; Xu, J.; Yu, J. X. (2004): Continuously maintaining quantile summaries of the most recent n elements over a data stream. *Data Engineering*, pp. 362-373.

Manku, G. S.; Rajagopalan, S.; Lindsay, B. G. (1998): Approximate medians and other quantiles in one pass and with limited memory. *SIGMOD Record*, vol. 27, no. 2, pp. 426-435.

Miao, D.; Liu, L.; Xu, R.; Panneerselvam, J.; Wu, Y. et al. (2018): An efficient indexing model for the fog layer of industrial internet of things. *Transactions on Industrial Informatics*, vol. 14, pp. 4487-4496.

Munro, J. I.; Paterson, M. S. (1978): Selection and sorting with limited storage. *Foundations of Computer Science*, pp. 253-258.

Papapetrou, O.; Garofalakis, M.; Deligiannakis, A. (2012): Sketch-based querying of distributed sliding-window data streams. *Proceedings of the Very Large Data Bases Endowment*, vol. 5, no. 10, pp. 992-1003.

Tangwongsan, K.; Hirzel, M.; Schneider, S. (2018): Sliding-window aggregation algorithms. *Proceedings of DEBS*, pp. 19-23.

Wang, H.; Zhang, Z.; Pengwei, W. (2018): A situation analysis method for specific domain based on multi-source data fusion. *Intelligent Computing Theories and Application*, pp. 161-171.

Wang, L.; Luo, G.; Yi, K.; Cormode, G. (2013): Quantiles over data streams: an experimental study. *Proceedings of the 2013 SIGMOD International Conference on Management of Data*, pp. 737-748.

Wu, Y.; Yan, C.; Liu, L.; Ding, Z.; Jiang, C. (2015): An adaptive multilevel indexing method for disaster service discovery. *Transactions on Computers*, vol. 64, no. 9, pp. 2447-2459.

Xu, J.; Zhang, D.; Liu, L.; Li, X. (2012): Dynamic authentication for cross-realm soa-based business processes. *Transactions on Services Computing*, vol. 5, no. 1, pp. 20-32.

Yu, C.-N.; Crouch, M.; Chen, R.; Sala, A. (2016): Online algorithm for approximate quantile queries on sliding windows. *International Symposium on Experimental Algorithms*, pp. 369-384.

Yu, W.; Ding, Z.; Liu, L.; Wang, X.; Crossley, R. D. (2018): Petri net-based methods for analyzing structural security in e-commerce business processes. *Future Generation Computer Systems*, pp. 10.

Zhang, X.; Zhang, Z.; Wang, L.; Zhou, X.; Wang, P. (2018): A novel method to improve hit rate for big data quick reading. *3rd International Conference on Computer Science and Information Engineering*, pp. 105-113.

Zhang, Z.; Cui, J. (2017): An agile perception method for behavior abnormality in large-scale network service systems. *Chinese Journal of Computers*, vol. 2, pp. 503-519.

Zhang, Z.; Ge, L.; Wang, P.; Zhou, X. (2017): Behavior reconstruction models for large-scale network service systems. *Peer-to-Peer Networking and Applications*, pp. 1-12.

Zhang, Z.; Zhou, X.; Zhang, X.; Wang, L.; Wang, P. (2018): A model based on convolutional neural network for online transaction fraud detection. *Security and Communication Networks*, vol. 2018, pp. 1-9.