Processing Sliding Windows over Disordered Streams

Hyeon Gyu Kim, Myoung Ho Kim

Department of Computer Science, Korea Advanced Institute of Science and Technology,
Guseong-dong, Yusong-gu, Daejeon, 305-701, South Korea
[hgkim, mhkim]@dbserver.kaist.ac.kr

Abstract—Bursty and out-of-order tuple arrivals complicate the process of determining the content and boundary of sliding windows. To process windows over such streams, two issues need to be addressed: how to sort input tuples efficiently and how to estimate punctuations. In this paper, we focus on these issues to process sliding windows efficiently and accurately over disordered streams. Regarding the first, we propose an order-preserving hash method to sort input tuples in constant time. Regarding the second, we present an estimation method based on the maximum distance between input tuples and their means. Based on these proposed methods, we finally provide a structure of window operators.

I. INTRODUCTION

There has been substantial research in processing continuous queries over unbounded data streams [1, 2, 3]. In continuous queries, sliding windows [4, 5] are usually employed to decompose unbounded streams into finite subsets. Stream decomposition is important in continuous queries because it enables blocking operators [2] to produce their results, which are unable to start processing until an entire input is seen (e.g. joins and aggregates).

To determine the contents and boundaries of sliding windows clearly, window operators generally assume that stream tuples arrive in an increasing order of a windowing attribute such as a generation timestamp of tuples [4]. However, stream tuples may not arrive in the order due to various sources of disorder such as network transmission delays, merging unsynchronized streams, data prioritization, and so on. To process sliding windows over disordered streams, two issues need to be addressed.

1. How to sort input tuples efficiently: Out-of-order tuples may lead to inefficiency and inaccuracy in window processing. Sorting tuples has to be conducted in an efficient manner to give real-time responses over bursty tuple arrivals.

2. How to estimate punctuations: To process a window with an interval of \( (t - n, t] \) where \( n \) denotes a window size and \( t \) a certain value, we have to guarantee that no more tuples \( \leq t \) will be seen in the stream. Such an assertion is referred to as a punctuation [6, 7] in the literature.

Regarding the first, existing approaches sort input tuples, which takes logarithmic or square time (e.g. Aurora [3] uses a bubble sort.) But we see that the sorting can be conducted in constant time by using a hash method. Regarding the second, existing approaches use an estimation method based on the maximum network delay [8, 9, 10]. But the method forces all input tuples to have their generation timestamps for deducing network delays of the tuples. From this restriction, it can be used only for timestamp-based sliding windows [5].

In this paper, we address these issues to process sliding windows efficiently and accurately over disordered streams. To sort tuples efficiently, we adopt an idea of the order-preserving hash method [11]. In our method, we divide a domain of input tuples into equal-sized disjoint segments; a segment corresponds to a hash entry and input tuples are mapped to hash entries by an order-preserving hash function (Figure 1). There are two notable things in our method.

1. Hash entries are always in an increasing order, while input tuples in a hash bucket may not be in the order. The necessity of sorting individual tuples in the bucket is determined by a given window specification. We provide running examples to illustrate which types of windows require sorting within the bucket, and give a brief mention about how to process each window types (Section 2).

2. The segment length is closely coupled with efficiency. We define a cost measure to compare feasible solutions and discuss an optimal length of segments with regard to the measure (Section 2).

To estimate punctuations, we propose a method based on the maximum value of distances between input tuples and their means (Section 3). Our estimation method doesn't have any restriction: the means can be obtained only by monitoring input tuples. Thus, its usage is not restricted to certain windows and to cover certain causes of disorder. We have verified our estimation method by conducting experiments.

Based on these methods described above, we propose a structure of window operators (Section 4), and then finally conclude our paper (Section 5). We focus on main memory processing and returning exact answers of given queries.

II. PROCESSING WINDOWS

This section presents running examples to illustrate how to process sliding windows over disordered streams. Via these examples, we elaborate two issues - inefficiency and
inaccuracy - occurring from disordered streams and give solutions based on our method. Then, we discuss the optimal length of segments which makes the processing cost to be minimal.

To specify sliding windows, we exploit the syntax proposed in [4]. A window specification consists of RANGE, SLIDE and WATTR, which denote the length of the window, the step by which the window moves, and the windowing attribute (i.e. the attribute over which RANGE and SLIDE are specified), respectively. A window specification \( w \) is defined as

\[
w = [\text{RANGE } R M_s, \text{SLIDE } S M_s, \text{WATTR } A]
\]

where \( R \) and \( S \) are integers, \( A \) is the name of an attribute, and \( M_s \) and \( M_s \) are the units of \( R \) and \( S \) such as "secs", "rows", etc. For example, consider a query "compute the maximum value of temperatures transmitted from environmental sensors over the past 10 seconds; update the value every 10 seconds.” This query can be written as shown in Q1. Below, we assume that the schema of sensor readings is \( <ts, \text{temp}> \), where \( ts \) denotes the timestamp given by sensor nodes and \( \text{temp} \) the temperature.

**Q1:**

\[
\text{SELECT MAX(temp) FROM Sensors [RANGE 10 secs, SLIDE 10 secs, WATTR ts]}
\]

In the processing perspective, we classify sliding windows into 3 groups according to \( M_s \) and \( M_s \): interval, count and mixed windows. In interval windows, \( M_s \) and \( M_s \) are an interval such as a time interval (e.g. secs, mins, etc). In count windows, both are a count such as tuple counts (e.g. rows). In mixed windows, \( M_s \) and \( M_s \) are different. For example, Q1 defines an interval window.

Let us first consider how to process interval windows. Figure 2 shows a logical query plan to process Q1, which consists of a window operator and a MAX (aggregate) operator. The window operator generates windows, each of which maintains input tuples within a range of 10 seconds. Its processing can be started when a punctuation is detected. In Figure 2, the punctuation \( p_l \) denotes that no more tuples with \( ts \leq 149 \) will be seen in the stream. When receiving \( p_l \), the window operator processes windows by accessing only tuples with \( ts \leq 149 \). It adds a delimiter (e.g. \( dl_1, dl_2 \) and \( d3 \)) to each window, which helps the successive operators to identify a boundary of the window. The MAX operator receives windows and generates the maximum value per window (e.g. \( m1, m2 \) and \( m3 \)).

Whenever processing a window over disordered streams, the window operator must scan all tuples maintained in its input queue to determine which tuples belong to the window. Such scanning leads to inefficiency in the processing because some tuples can be scanned multiple times. To resolve this, existing approaches sort individual tuples. For example, Aurora uses a bubble sort for this purpose.

However, sorting tuples can be conducted in constant time by using our order-preserving hash method. For example, to process windows given in Q1, we maintain a list of buckets, each of which covers an interval of 10 seconds (Figure 3). A simple hash function (e.g. \(<ts> / 10\)) is used to map input tuples to their corresponding hash entries. Note that sorting tuples in each bucket is not necessary. A window can be made from concatenating successive buckets. Number of buckets in a window can be calculated as \( R / L \), where \( R \) is a window size (i.e. the value of RANGE) and \( L \) is a segment length.

Now, let’s turn to the case of mixed windows; we do not consider how to process count windows, because it is straightforward since there is no disorder in that type of windows. Consider the following query Q2, where windows are slid by every 2 tuples.

**Q2:**

\[
\text{SELECT MAX(temp) FROM Sensors [RANGE 10 secs, SLIDE 2 rows, WATTR ts]}
\]
Figure 4 shows a logical query plan to process Q2, which also consists of a window operator and a MAX operator. In this case, the behavior of the window operator is different from that in Q1: given a window specification with SLIDE of $S$ rows, the interval of each window is determined by every $S$ tuples. In Figure 4, the interval of the first window $w1$ is determined by the first tuple $t1$, the interval of $w2$ by $t5$, the interval of $w3$ by $t3$, and so on.

When processing mixed windows, there is also an inefficiency issue because the whole scan of the input queue is still required to process one window. But there is more significant issue in this case - inaccuracy. Some tuples may occur more than one time in generated windows and some tuples may not occur in any windows. For example, in Figure 4, $t5$ occurs in both $w2$ and $w3$, and $t6$ does not occur in any windows. In addition, the generated windows may not be in an increasing order. If $ts$ of $t3$ is changed to 115, the interval of $w2$ becomes $[115, 124]$ that is smaller than the interval of $w1$.

When using our method in this case, there still remain inefficiency and inaccuracy issues. For example, consider the status of the window operator after applying our method (Figure 3). To process $w2$ whose interval $[135, 144]$ is determined by $t3$, two buckets of intervals $[130, 139]$ and $[140, 149]$ have to be scanned. In addition, $t6$ still occurs in both $w2$ and $w3$, and $t6$ does not occur in any windows. To resolve all of these issues in mixed windows, the tuples in each bucket have to be sorted.

If we have information about an average arrival rate of input tuples, it is possible to avoid the sorting in the buckets by choosing an appropriate length of segments. To discuss feasible solutions for the segment length, we define a cost measure as follows.

\[ C = \# \text{ of tuple accesses for processing one window} \]

Let $\lambda$ be an average arrival rate of input tuples and $N$ be the number of segments in a window. Then, the number of tuples in a window can be calculated as $R \cdot \lambda$ and the number of tuples in a segment as $R \cdot A / N$. The processing cost consists of i) the cost for sorting the tuples in each segment of a window and ii) the cost for accessing tuples to present them as a window, which can be represented as

\[ C = N \cdot (R \cdot A / N) \cdot \log(R \cdot A / N) + (R \cdot A) \]

Intuitively, the cost of (5) is minimal when the sorting cost is excluded. The following theorem gives optimal length of segments in mixed windows, which coincides with our intuition.

**Theorem 1.** The optimal size of segments with regard of $C$ is $1$ if $\lambda \geq 1$ or $1 / \lambda$ otherwise.

**Proof.** We first consider the case of $\lambda \geq 1$. Let $L$ be the segment length, which is equal to $R / N$. Then, we can rewrite the above cost formula by substituting $N$ with $R / L$.

\[ C = (R \cdot \lambda) \cdot \log(L \cdot \lambda) + (R \cdot A) \]

In above expression, $R$ and $\lambda$ are the given constants, Then $C$ becomes minimal when $L$ is 1.

Now, we discuss the case of $\lambda < 1$. In this case, $C$ is no longer valid because it may give a negative value. If we choose 1 as the segment size, the array of segments can be sparse, which leads to unnecessary memory consumption. If we choose $1 / \lambda$ as the segment size, each segment has 1 tuple on the average, so it is possible to avoid the memory utilization issue. If we choose a larger value than $1 / \lambda$, each segment has more than 1 tuples, which incurs sorting in the segments. \[ \square \]

### III. ESTIMATING PUNCTUATIONS

Existing approaches usually estimate punctuations based on the maximum network delay. The network delay of an input tuple can be obtained by its arrival timestamp minus its generation timestamp. To estimate punctuations, it maintains
method
required
tupledrops,u.
that
disordercontrol
becomes
exceed
valueofthedistancesof
inputtuples
applicationrequirements.

Notethat
We
usesan
support
evaluation,execution,

Ourmethodmaintainsthemaximum
tuple.
Ourmethod
cover
any
inputtuples
ofthelatest
tuple.

Fig.5 Our method vs. delay-based method - (a) buffer sizes and (b) drop ratios

the maximum value of the network delays of tuples in the streams seen so far. A punctuation is calculated as the current system timestamp minus the maximum delay. We refer to this method as a delay-based method.

There are some restrictions in this method. First, it only focuses on disorder resulting from network delays and cannot cover any other causes of disorder. Second, it forces every input tuple to have their generation timestamp, which is required to deduce network delays. From this restriction, this method can be applied only to timestamp-based windows. Finally, there is no way to control disorder. Users may want tuple drops not to exceed some predefined limit from application requirements.

Our method estimates punctuations based on the maximum distance between input tuples and their means. A distance of an input tuple can be obtained by a difference between a mean of the latest input tuples and a value (of the windowing attribute) of the tuple. Our method maintains the maximum value of the distances of tuples in the streams seen so far. A punctuation is calculated as the mean μ minus the maximum distance. We currently monitor the latest λ tuples to calculate μ. We refer to our method as a mean-based method.

Note that our method doesn’t have any restriction: μ can be obtained by monitoring input tuples. Thus, its usage is not limited to certain windows and to cover certain causes of disorder. In addition, our method enables users to control tuple drops occurring from the estimation process. To support the disorder control in a declarative manner, we define an optional parameter DRATIO, which denotes a percentage of tuple drops permissible during query execution.

Q3: SELECT MAX(temp) FROM Sensors [RANGE 10 secs, SLIDE 10 secs, WAITR TS DRATIO 1%]

To support DRATIO, we currently use a heuristic approach that uses an offset value to adjust the estimated punctuation to be delayed or forwarded. For example, if a resulting drop ratio becomes larger (or smaller) than the given DRATIO during query execution, we increase (or decrease) the offset value to delay (or forward) next punctuations. The offset value is refreshed every λ tuple arrivals in our approach.

We have conducted two experiments to test accuracy of our estimation method. We implemented a data generator to synthesize data sets based on given input parameters such as a tuple arrival rate, a standard deviation of disorder, the maximum disorder, number of tuples, and others. Our experiments were conducted on Intel Pentium 4 2.4 MHz machine, running Window XP, with 1 G main memory.

The first compares the delay-based method and our method in terms of buffer sizes and drop ratios. We had two notable results: i) the resulting buffer sizes of both methods converge to a same value as a data size increases (Figure 5 (a)), and ii) our method shows smaller tuple drops during the estimation (Figure 5 (b)). The second checks whether estimation results observe a drop ratio given from a query specification. The results show that our method satisfies accuracy; it does not violate any given drop ratios (Figure 6). These results show that our method can substitute the delay-based method without any deterioration of accuracy, while providing a method for disorder control.

IV. STRUCTURE OF WINDOW OPERATORS

In the previous sections, we provide methods to process sliding windows over disordered streams. With these methods, we propose a structure of window operators as shown in Figure 7, which consists of three major components: record
store, disorder controller and window processor. Record store uses an order-preserving hash function to order input tuples in constant time and maintains an array of buckets to store the tuples. Disorder controller estimates punctuations based on the maximum distance between input tuples and their means, which method can be used without any restriction. Window processor generates sliding windows by concatenating tuples in record store, whenever a punctuation arrives from disorder controller.

V. CONCLUSIONS

In this paper, we proposed methods to process sliding windows efficiently and accurately over disordered streams. First, we proposed an order-preserving hash method to sort out-of-ordered input tuples in constant time. Then, we presented an estimation method that can be used without any restriction; its usage is not restricted to certain windows and to cover certain causes of disorder. Finally, we provided a structure of window operators based on these methods.

ACKNOWLEDGMENT

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korean government (MOST) (No. R0A-2007-000-10046-0).

REFERENCES