LETTER

Efficient Window Processing over Disordered Data Streams

Hyeon-Gyu KIM[†a)], ***Member***, **Woo-Lam KANG**[†], ***and*** **Myoung-Ho KIM**[†], ***Nonmembers***

SUMMARY Bursty and out-of-order tuple arrivals complicate the process of determining contents and boundaries of sliding windows. To process windows over such streams efficiently, we need to address two issues regarding fast tuple insertion and disorder control. In this paper, we focus on these issues to process sliding windows efficiently over disordered data streams.

key words: *data streams, window processing, disorder control*

1. Introduction

Continuous query processing over data streams has attracted a lot of interest recently. Since data streams are so large or often inherently unbounded, queries on the streams cannot be easily answered if they involve *blocking operators* such as joins or aggregations which operators cannot start processing until the entire inputs are ready. A common solution for this issue is to restrict the range of stream queries into a *sliding window* that contains the most recent data of the stream [1], [2].

For example, consider a query that asks for the maximum value of sensor readings over the latest 30 seconds. This query can be specified as $Q1$ which is a SQL-like query with window specification "[RANGE 30 seconds]". The query is manipulated for each tuple arrival. Whenever a new tuple with timestamp t arrives, a window operator in a stream query processor, also called a *DSMS* (Data Stream Management System), determines the window interval by $(t - 30, t]$ and organizes a collection of tuples belonging to the window which is called a *window extent* [6]. Then, the aggregate function *MAX* finds the maximum sensing value among the tuples in the window extent.

```
Q1. SELECT MAX(value)
    FROM Sensors [RANGE 30 seconds]
```

To determine contents and boundaries of sliding windows clearly, window operators generally assume that stream tuples arrive in an increasing order of the *windowing attribute* - an attribute by which window ranges are determined (e.g., tuples' generation timestamps). 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. These out-of-order tuples may lead to inaccurate query results since conventional window operators usually discard those tuples.

Existing approaches use a buffer to fix disorder in stream tuples before transferring those tuples to window operators. They generally focus on avoiding tuple discards as many as possible. Consequently, the approaches tend to keep the buffer size larger than necessity. On the other hand, users may want to have faster results (with less accuracy) according to application requirements. Such faster results can be provided by adjusting the buffer size properly (e.g., smaller). However, existing approaches have not addressed this issue even though a method for controlling disorder is necessary in processing data streams as discussed in [6], [7].

To process sliding windows efficiently over disordered streams, we also need a mechanism for fast insertion of input tuples while keeping their windowing attribute order in the buffer. *Aurora* [1] which is one of the well-known DSMSs uses a bubble sort to order incoming tuples. But, if the number of tuples in the buffer becomes large (due to bursty tuple arrivals), conventional sorting mechanisms with logarithmic complexity may not be feasible in providing query responses in a timely manner.

In this paper, we propose a method for processing sliding windows efficiently over disordered data streams, regarding two issues described above - (i) fast tuple insertion and (ii) disorder control.

2. Backgrounds

There has been substantial research in the problem of processing disordered data streams. *Aurora* [1] uses a fixed-size buffer to deal with disorder in stream tuples. Assuming that the max delay is known in advance, it uses a buffer whose size is large enough to cover the max delay. If we don't have any prior knowledge about the max delay, the fixed-size buffer may not be used properly because it is hard to decide the buffer size; too small buffer may cause many tuples to be discarded, while too large buffer results in high latency because input tuples reside in the buffer long time.

Many other DSMSs including *STREAM* [3] use a dynamic buffer. When using the buffer, we need to identify which tuples can be outputted from the buffer at a certain time; otherwise, the buffer will grow infinitely without any output. Existing approaches generally determine those tuples based on the *maximum network delay seen in the stream*. Let the max delay in the stream be m. Then, at

Manuscript received November 5, 2009.

[†]The authors are with the School of Electrical Engineering and Computer Science, the Division of Computer Science, KAIST, 373–1 Kusong-dong, Yusong-gu, Daejeon 305–701, South Korea.

a) E-mail: hgkim@dbserver.kaist.ac.kr

DOI: 10.1587/transinf.E93.D.635

time t, the buffer keeps tuples whose timestamps are larger than $t - m$, and other tuples in the buffer are delivered to a window operator. The *k-ordering* mechanism [3], the *skew bound* estimation in [8], the timestamp mechanism in *Gigascope* [5] are similar to this approach.

Note that if a new tuple with timestamp smaller than $t - m$ arrives after sending out those tuples (with timestamps $\leq t - m$), it is discarded from the buffer. The timestamp $t - m$ is called a *heartbeat* [5], [8] or a *punctuation* [6] in the literature. The heartbeat h is an assertion indicating that no more tuples with timestamps smaller than h will be seen in the future. Therefore, given heartbeat h, it is possible to process tuples with timestamps smaller than h. If new tuples with values smaller than h arrive after that processing, they will be discarded.

The heartbeat can be estimated internally by DSMSs or can be given externally from other remote stream sources such as routers. Ding et al. [4] and Jin Li et al. [6] assume that the heartbeats are externally given and propose methods for processing join or aggregate queries gracefully based on the given heartbeats. However, external stream sources may not provide heartbeats in real-world applications. In addition, the heartbeats themselves can be out-of-ordered when the stream sources are in remote locations.

When estimating the heartbeat internally, existing approaches generally focus on avoiding tuple discards as many as possible. Consequently, they tend to keep the buffer size larger than necessity. For example, the *adaptive method* proposed by Srivastava et al. [8] estimates the max delay m by $(m_1 + m_2)/2$, where m_1 is the max delay seen in the stream at time t and m_2 is the second max delay in a time interval $[t, t + W]$ (Fig. 1). We can easily see that m is kept large in most of time since it is determined by two largest values of network delays seen in the stream in a certain period of time. Moreover, there is no explicit method to control the amount of tuple discards in this method. To control tuple drops, we need to adjust W properly, but they did not discuss how to determine it.

Recently, Liu et al. [7] proposed two out-of-order handling techniques, *conservative and aggressive strategies*, to enable users to control the quality of query results according to application requirements. The conservative strategy can be used to avoid tuple discards as many as possible to obtain accurate query results. On the other hand, the aggressive strategy is targeted to provide faster results. However, they do not provide a fine-grained control. In their method, users can simply choose one of the two choices and cannot control the amount of tuple discards in the level of a "percentage"; the percentage of tuples delivered (or discarded) is one of the popularly used QoS (Quality of Service) parameters in DSMSs [1].

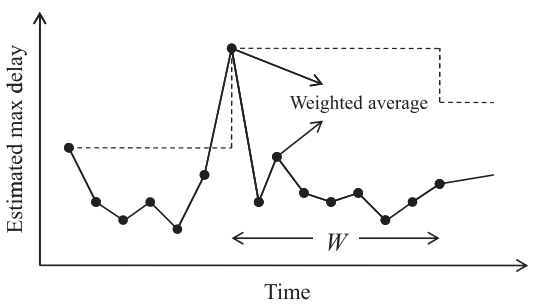

Fig. 1 Estimation of the max delay in the adaptive method proposed by Srivastava et al. [8].

3. Window Processing

In this section, we discuss how to process sliding windows efficiently over disordered data streams. To simplify our discussion, we only focus on the processing of time-based windows whose ranges are determined by tuples' generation timestamps. Extension to the processing of other windows is straight-forward and we will briefly discuss it if needed.

Our method uses a dynamic buffer to fix disorder of stream tuples and estimates a heartbeat to determine tuples that can be outputted from the buffer. We generate window extents based on the output tuples. Figure 2 shows the structure of our window operator which consists of three main parts: the *record store*, the *disorder controller* and the *window generator*. The record store receives input tuples and provides a pool of tuples over which the window generator produces window extents. The disorder controller estimates the heartbeat by monitoring input tuples.

The record store again consists of the *slack buffer* and the *window buffer*. The slack buffer includes input tuples whose timestamps are larger than the current heartbeat h, while the window buffer has tuples whose timestamps are smaller than or equal to h. The slack buffer needs to be organized to accept input tuples efficiently while keeping their timestamp order. The buffer size may significantly grow due to bursty tuple arrivals. In this case, ordering tuples based on conventional sorting mechanisms with logarithmic complexity can be an exhausting job.

Fortunately, we can handle each incoming tuple in constant time. Many DSMSs assume that tuples' timestamps are values from a discrete, ordered time domain such that the values can be represented as nonnegative integers starting from zero (e.g., seconds) [2]. Based on this assumption, we can easily group input tuples according to their times-

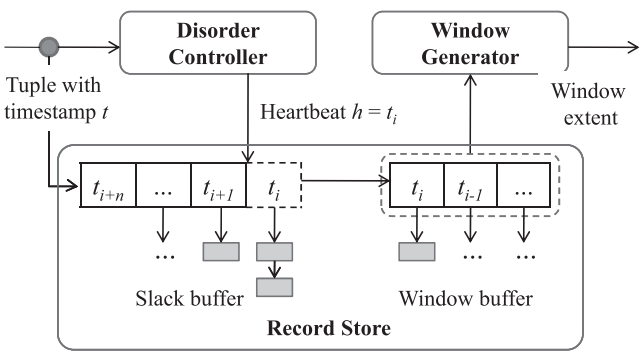

Fig. 2 Structure of our window operator.

tamps by using an array whose elements correspond to tuples' timestamps. Actually, we organize the slack buffer as a circular array whose element has a set of tuples with the same timestamp. Then, we map an input tuple to one of elements directly by array indexing based on the tuple's timestamp. The window buffer has the same structure of the slack buffer.

Whenever a new tuple arrives in the buffer, we first check its windowing attribute value to determine whether we accept it or not. The determination is conducted based on the heartbeat h estimated by the disorder controller. If the tuple's timestamp t is smaller than h, the tuple is discarded. If t is equal to h, we directly output the tuple to the window buffer. Otherwise, we keep it in the slack buffer.

Tuples in the slack buffer are delivered to the window buffer when the heartbeat h increases than before. Given a new heartbeat h', we output tuples with timestamp smaller than h' to the window buffer. Note that the heartbeat must monotonically increase, which is required to guarantee the timestamp order of tuples outputted from the slack buffer. The disorder controller may estimate the new heartbeat h' whose value is smaller than the previous one h. In this case, we keep h' to be equal to h.

4. Heartbeat Estimation

As discussed earlier, existing approaches generally estimate the heartbeat based on the maximum network delay seen in data streams. The network delay of the i-th input tuple s_i can be calculated by u_i - t_i where u_i is an arrival time of s_i and t_i is its generation timestamp. Then, the approaches keep the maximum value of network delays seen so far. Given the max delay m, the heartbeat at the arrival of s_i is estimated by $u_i - m$.

Note that the method can be used when input tuples have their generation timestamps and can be applied to time-based windows only. It cannot be used for other types of windows whose windowing attributes are not generation timestamps. For example, consider a query that asks an average service time of the latest 100 orders which can be served by different stores located in different places. The query can be specified as $Q2$, where *Services* denotes an input stream that unions service results relayed from different stores. Each result consists of < *orderID*, *serviceTime*, ... > where *orderID* is an ID issued when the order is received and *serviceTime* is a period of time between reception and completion of the order. We assume that the orders are sequentially issued in one cite to which their results are also relayed.

```
Q2. SELECT AVG(serviceTime)
    FROM Services [RANGE 100 tuples,
                   WATTR orderID]
```

Since orders are served in different places, the arrivals of their results can be out-of-ordered. But, we cannot use the existing method to fix disorder of the stream *Services* because the windowing is not conducted based on tuples' timestamps; the windowing attribute is *orderID* in this case. Moreover, there is no way to control disorder of stream tuples in the method. According to application requirements, users may want to control the amount of tuple discards not to exceed a predefined bound.

To resolve these issues, we use the *mean of tuples' windowing attribute values* for the heartbeat estimation. More specifically, whenever a tuple s_i arrives, we calculate its distance from the current mean μ by $|a_i - \mu|$, where a_i is the windowing attribute value of s_i. Similar to the existing method, we keep the maximum value of tuples' distances seen in the streams so far. Given the maximum distance d, the heartbeat at the arrival of s_i is estimated by $\mu - d$.

In this way, our method doesn't have any restriction: μ can be obtained simply by monitoring values of the windowing attribute of input tuples. Therefore, its usage is not limited to certain types of windows, and it can cover any causes of disorder including tuples' transmission delays, merging unsynchronized streams (as in the case of $Q2$), and so on.

The proposed method also enables users to control the amount of tuple discards occurring in the estimation process. To support disorder control, we provide an optional parameter *DRATIO* which denotes a percentage of tuple drops permissible during the query execution. By specifying the parameter, users can control the quality of query results according to application requirements; a small value of the drop ratio provides more accurate query results at the expense of high latency, while a large value gives faster results with less accuracy.

```
Q3. SELECT AVG(serviceTime)
    FROM Services [RANGE 100 tuples,
                   WATTR orderID,
                   DRATIO 1%]
```

To satisfy DRATIO, we currently use a heuristic method. We use an offset value to adjust the estimated heartbeat to be delayed or forwarded. For example, if a drop ratio shown in the estimation process becomes greater than the specified DRATIO, we increase the offset to delay the heartbeat. On the other hand, if the drop ratio is less than the specification, we decrease it to forward the heartbeat value. The offset is refreshed periodically in our method (e.g., every second).

We have conducted two experiments to see availability of our estimation method. We implemented a data generator to synthesize data sets based on user-specified parameters including an arrival rate of input tuples, a standard deviation of disorder, the maximum value of disorder, and so on. Our experiments were conducted on Intel Pentium IV 2.4 MHz machine, running Window XP, with 1 G main memory.

The first compares our method and the existing method based on the max delay [8] in terms of buffer sizes and drop ratios. For this experiment, we set DRATIO to 0% which value can be used to save tuple discards as many as possible (i.e., the best effort strategy). We had two results: the buffer sizes of both methods converge to the same value as the data size increases (Fig. 3 (a)) and our method shows smaller

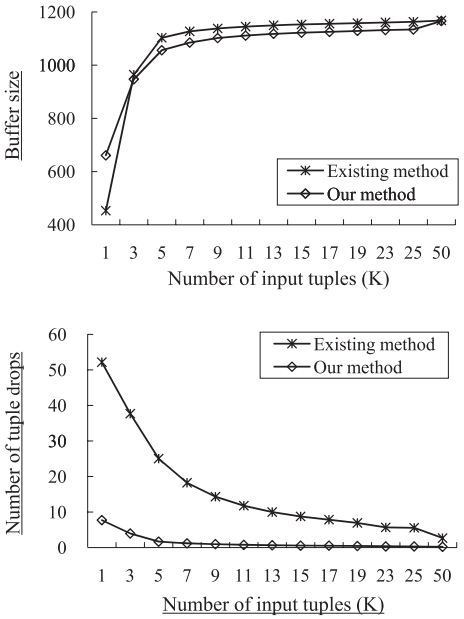

Fig. 3 Existing vs. our methods: (a) buffer size (upper) and (b) number of tuple drops (below).

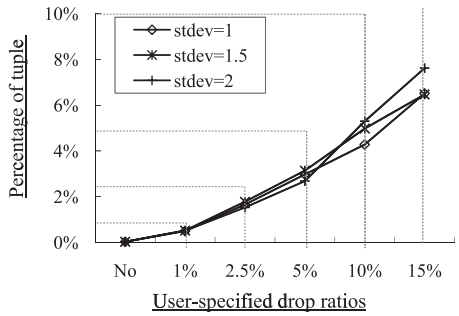

Fig. 4 Our method: Specified drop ratios vs. percentages of tuple drops.

tuple drops during the estimation (Fig. 3 (b)). The second checks whether our method observes a user-specified drop ratio. The results in Fig. 4 show that it does not violate the given drop ratios. These results show that our method can be used instead of the existing method without deterioration of accuracy while providing a method for disorder control.

5. Conclusion

In this paper, we proposed a method to process sliding windows efficiently over disordered data streams. We first proposed the structure and algorithm for processing windows over disordered streams. Then, we provided an estimation method that can substitute the existing method without deterioration of accuracy while providing a method for disorder control, which we observed through our experiments.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2009-0083055).

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," VLDB Journal, vol.12, no.2, pp.120–139, 2003.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," Proc. ACM PODS 2002, pp.1–16, Madison, Wisconsin, United States, 2002.

[3] S. Babu, U. Srivastava, and J. Widom, "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams," ACM Trans. Database Syst. (TODS), vol.29, no.3, pp.545–580, 2004.

[4] L. Ding and E.A. Rundensteiner, "Evaluating window joins over punctuated streams," Proc. ACM CIKM 2004, pp.98–107, Washington, DC, United States, Nov. 2004.

[5] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck, "A heartbeat mechanism and its application in gigascope," Proc. VLDB Conf. 2005, pp.1079–1088, Trondheim, Norway, Sept. 2005.

[6] Jin Li, D. Maier, K. Tufte, V. Papadimos, and P.A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," Proc. ACM SIGMOD 2005, pp.311–322, Baltimore, Maryland, United States, June 2005.

[7] M. Wei, M. Liu, M. Li, and K. Claypool, "Supporting a spectrum of out-of-order event processing technologies: From aggressive to conservative methodologies," Proc. ACM SIGMOD 2009, pp.1031–1033, Providence, Rhode Island, United States, June 2009.

[8] U. Srivastava and J. Widom, "Flexible time management in data stream systems," Proc. ACM PODS 2004, pp.263–274, Paris, France, June 2004.