Computing and Informatics, Vol. 31, 2012, 393–410

ADAPTIVE DISORDER CONTROL IN DATA STREAM PROCESSING

Hyeon Gyu Kim

Integrated Safety Assessment Division
Korea Atomic Energy Research Institute
e-mail: hgkim@dbserver.kaist.ac.kr

Cheolgi Kim*

School of EECS
Korea Aerospace University
e-mail: cheolgi@illinois.edu

Myoung Ho Kim

School of Electrical Engineering and Computer Science
Division of Computer Science
Korea Advanced Institute of Science and Technology
373-1 Guseong-Dong, Yuseong-Gu, Daejeon 305-701, South Korea
e-mail: mhkim@dbserver.kaist.ac.kr

Communicated by P. K. Mahanti

Abstract. Out-of-order tuples in continuous data streams may cause inaccurate query results since conventional window operators generally discard those tuples. Existing approaches use a buffer to fix disorder in stream tuples and estimate its size based on the maximum network delay seen in the streams. However, they do not provide a method to control the amount of tuples that are not saved and discarded from the buffer, although users may want to keep it within a predefined error bound according to application requirements. In this paper, we propose a method to estimate the buffer size while keeping the percentage of tuple drops

* corresponding author

within a user-specified bound. The proposed method utilizes tuples' interarrival times and their network delays for estimation, whose parameters reflect real-time stream characteristics properly. Based on two parameters, our method controls the amount of tuple drops adaptively in accordance with fluctuated stream characteristics and keeps their percentage within a given bound, which we observed through our experiments.

Keywords: Data stream processing, sliding windows, buffer estimation, disorder control, drop ratio

1 INTRODUCTION

Recent years have witnessed the emergence of a new class of applications, which monitor continuous streams of data items such as stock exchanges, network measurements, web page visits, sensor readings and so on [1, 2, 3]. Since these 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; these are the operators which 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 [2, 3, 4].

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. Then, the aggregate function MAX finds the maximum sensing value among the tuples in the window.

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

In general, window operators assume that tuples arrive in an increasing order of their timestamps and discard out-of-order tuples to simplify their processing [5]. However, tuples may not arrive in the order since they often experience different network transmission delays when a stream source is in a remote location. These out-of-order tuples may lead to inaccurate results in an aggregate query such as $Q1$.

Existing approaches use a buffer to fix disorder of stream tuples before they are inputted to window operators. The size of the buffer can be fixed or variable. The fixed-size buffer is used in *Aurora* [1] which is one of the well-known DSMSs; but, such a buffer cannot be used properly when information about stream characteristics (e.g., network delays) is not available in advance, because it is hard to decide the buffer size in this case. Too large buffer results in large latency because input tuples

reside in the buffer long time. On the other hand, too small buffer may cause many tuples to be discarded since a new tuple is discarded when its timestamp is smaller than that of the latest tuple outputted from the buffer.

Many other DSMSs including *STREAM* [6] 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 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. If a new tuple with timestamp smaller than $t - m$ arrives after that, it is discarded from the buffer.

Note that in the existing approaches, there is no explicit method to control the amount of tuple discards. The feature for disorder control is important because users may want to force the percentage of tuple drops not to exceed a predefined bound according to application requirements. This feature becomes more useful when the memory is run out from a huge size of the buffer caused by bursty tuple arrivals. When the memory is not enough, DSMSs generally discard some tuples from the memory to shed system load. To maximize the effect of load shedding, they try to drop tuples as in the early stage of query processing as possible. At the same time, they need to control the amount of tuple discards to meet the QoS (Quality of Service) specification given by a user, which describe the criteria about the percentage of tuples delivered or discarded [1, 7, 8]. These lead to the necessity of disorder control in window processing.

In this paper, we propose a method to estimate the buffer size while keeping the percentage of tuple drops within a user-specified bound. We provide an optional parameter *DRATIO* (acronym of a *drop ratio*) that can be described in a window specification as shown in $Q2$ below. 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 (due to a large buffer), while a large value gives faster results with less accuracy (from a smaller buffer).

```
Q2. SELECT MAX(value)
    FROM Sensors [RANGE 30 seconds, DRATIO 1%]
```

We derive an estimation function based on a stream model where input tuples are randomly generated and arrive after normally-distributed network delays. The derived function utilizes tuples' interarrival times and their network delays for estimation, whose parameters reflect real-time stream characteristics properly. Based on two parameters, our method controls the amount of tuple drops adaptively in accordance with fluctuated stream characteristics and observes a given drop ratio. We show through our experimental results that

- it is hard to control the amount of tuple drops in the existing method which adjusts the buffer size heuristically, and

- the proposed method observes a user-specified drop ratio in processing data streams whose characteristics are dynamically changed.

The rest of this paper is organized as follows. Section 2 introduces existing work for processing windows over disordered streams. Section 3 proposes our method to estimate the buffer size adaptively based on the two run-time parameters. Section 4 provides our experimental results that compare our method with the existing method based on the max delay seen in the stream. Section 5 concludes our discussion with a brief mention of future work.

2 RELATED WORK

In general, DSMSs evaluate queries over potentially infinite streams of tuples. To produce outputs continuously when the queries involve blocking operators such as joins or aggregates, they are required to limit the scope of query processing to the most recent data of streams. Conventional DSMSs including *Aurora* [1], *STREAM* [6] and *TelegraphCQ* [9] commonly use sliding windows to localize the scope of input tuples.

Many types of windows have been proposed so far, including *time-based windows*, *tuple-based windows*, *value-based windows* [2], *predicate windows* [10] and so on. Among them, time-based windows are most frequently used for continuous queries [11, 12], whose window intervals are determined by tuples' timestamps. However, their boundaries and contents may not be easily identified when incoming tuples are not in the timestamp order [5, 13].

To resolve this issue, Aurora 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 do not 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 large latency because input tuples reside in the buffer for a long time.

Many other DSMSs including STREAM use a dynamic buffer to fix disorder of input tuples. As discussed earlier, they usually determine the tuples that can be outputted from the buffer, based on the max delay seen in the stream. Given the max delay m at 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 [14], the *skew bound* estimation in [12], the timestamp mechanism in *Gigascope* [15, 16] and the ordering mechanism in *NiagaraCQ* [17] are similar to this approach.

The timestamp $t - m$ is called a *heartbeat* [12, 16] or a *punctuation* [5, 13, 18] 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. [19] and Jin Li et al. [5] 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.

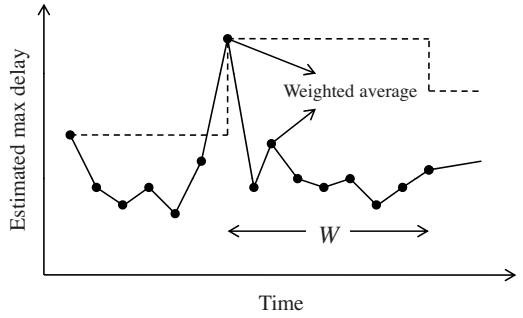


Fig. 1. Estimation of the max delay in the adaptive method proposed by Srivastava et al. [12]

When estimating the heartbeat internally, existing approaches generally focus on saving discarded tuples as many as possible. Consequently, they tend to keep the buffer size larger than necessary. For example, the *adaptive method* proposed by Srivastava et al. [12] 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 $[t, t+W]$ (Figure 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 the existing approaches. To control the tuple drops, we need to adjust the parameter W according to fluctuated stream characteristics. However, the adaptive method did not discuss how to determine W properly. It is hard to expect that the method based on a heuristically determined W will observe a user-specified bound.

3 OUR METHOD

In this section, we first introduce a stream model that characterizes tuple generations and arrivals in our method. Then, we derive an equation to estimate a buffer size based on the stream model; the equation is developed to observe a user-specified drop ratio. Finally, we provide the structure and the algorithm to fix disorder of stream tuples using the buffer whose size is estimated by the derived equation.

3.1 Stream Model

In our stream model, times for tuple generations at stream sources are randomly distributed, and the number of tuples to be generated follows a *Poisson process* with parameter λ. In other words, their generation intervals have an exponential distribution with mean θ ($= 1/\lambda$). In what follows, T_i and U_i are random variables for timestamps that denote the generation time (given by a remote stream source) and the arrival time (stamped at a query processor) of the i-th arrival tuple, respectively (Figure 2).

$$(T_i - T_{i-1}) \backsim Exp(\theta) \tag{1}$$

Network delays experienced by tuples follow a normal distribution with mean μ and standard deviation σ.

$$(U_i - T_i) \backsim Norm(\mu, \sigma^2) \tag{2}$$

Then, interarrival times of the tuples follow an exponential distribution with mean θ, whose proof is given in Appendix.

$$(U_i - U_{i-1}) \backsim Exp(\theta) \tag{3}$$

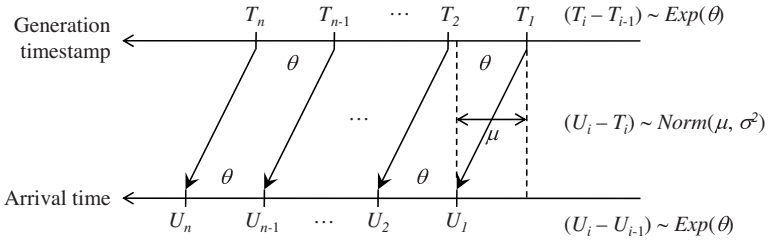


Fig. 2. Stream model that characterizes tuple generations and network delays in our method

Figure 2 depicts our stream model described above. In the figure, U_1 and U_n denote the arrival times of the earliest and the latest tuple in the buffer B, respectively, where n is the number of tuples in B. Similarly, T_1 and T_n are generation timestamps of the earliest and the latest tuples, respectively.

3.2 Estimation Function

As mentioned earlier, we support users to specify a *drop ratio* in a query. The drop ratio denotes a percentage of tuple discards permissible during the process of disorder control. A user may choose a small value of the drop ratio to get more accurate query results at the expense of a large buffer with high latency. On the

other hand, he/she can choose a large value to have faster results by using a smaller buffer, but with less accuracy.

The drop ratio can be specified by using an optional parameter *DRATIO* in our method. It can be augmented to a window specification. The query $Q3$ below shows an example where DRATIO is given to 1 %. It indicates that the percentage of tuple drops (i.e., number of tuple drops per total number of input tuples) should be less than or equal to 1 %.

```
Q3. SELECT MAX(value)
    FROM Sensors [RANGE 5 minutes, DRATIO 1%]
```

Now, let us derive an equation to estimate a buffer size which observes a drop ratio specified in a query. Without loss of generality, suppose there are n tuples s_1, s_2, ..., s_n in the buffer B. Then, the current heartbeat is less than T_1 by definition since T_1 is the smallest timestamp among those tuples. Consider a new tuple that does not yet arrive in B, whose timestamp is T_{n+1}. The condition that the tuple is discarded from B is $T_{n+1} < T_1$. Thus, the probability of a tuple drop can be represented as $Pr(T_{n+1} < T_1)$, and it must be smaller than a user-specified drop ratio D.

$$Pr(T_{n+1} - T_1 < 0) \leq D \tag{4}$$

$T_{n+1} - T_1$ can be rewritten as follows.

$$(T_{n+1} - T_1) = (T_{n+1} - U_{n+1}) + (U_{n+1} - U_1) + (U_1 - T_1) \tag{5}$$

For convenience, we use random variables S, X, Y and Z for $(T_{n+1} - T_1)$, $(T_{n+1} - U_{n+1})$, $(U_{n+1} - U_1)$ and $(U_1 - T_1)$, respectively.

Since X and Z are network delays, they have normal distributions from (2). We can also approximate Y to a normal distribution. Y can be interpreted as a sum of interarrival times of tuples, that is, $\sum_i (U_i - U_{i-1})$ $(1 \leq i \leq n+1)$. If the number of tuples n is greater than or equal to 30 (we will do so in our method), we can approximate the sum of $(U_i - U_{i-1})$ to a normal distribution from the *Central Limit Theorem* [20]. In this case, a mean and a standard deviation of the distribution become $n\theta$ and $n\theta^2$, respectively. As a result, all of X, Y and Z have normal distributions as follows.

$$X \backsim \text{Norm}(-\mu, \sigma^2) \tag{6}$$

$$Y \backsim \text{Norm}(n\theta, n\theta^2) \tag{7}$$

$$Z \backsim \text{Norm}(\mu, \sigma^2) \tag{8}$$

To get the distribution of S (i.e., the convolution of X, Y and Z), we use the MGFs (Moment Generating Functions) [20] of the three random variables that have normal distributions.

$$\begin{aligned} M_S(s) &= M_X(s) \cdot M_Y(s) \cdot M_Z(s) \\ &= e^{\sigma^2 s^2/2 - \mu s} \cdot e^{n\theta^2 s^2/2 + n\theta s} \cdot e^{\sigma^2 s^2/2 + \mu s} \\ &= e^{(\sigma^2 s^2/2 - \mu s) + (n\theta^2 s^2/2 + n\theta s) + (\sigma^2 s^2/2 + \mu s)} \\ &= e^{(2\sigma^2 + n\theta^2) s^2/2 + n\theta s} \end{aligned}$$

Note that the result is again in a form of the normal distribution MGF, where a mean and a standard deviation of the distribution are $(n\theta)$ and $(2\sigma^2 + n\theta^2)$, respectively. Therefore, the distribution of $(T_{n+1} - T_1)$ is

$$(T_{n+1} - T_1) \backsim \text{Norm}(n\theta, 2\sigma^2 + n\theta^2) \tag{9}$$

By standardizing (9), we can have a function to get the probability of a tuple drop as shown in the left expression of (10), where Φ is the cumulative distribution function of a standard normal distribution [20]. Based on the function, the condition (4) can be rewritten as

$$\Phi(\frac{n\theta}{\sqrt{2\sigma^2 + n\theta^2}}) \geq 1 - D. \tag{10}$$

We solve (10) as an equation and get a solution for n as shown in (11). Below, $z(\alpha)$ is a function that returns the *percentile* (also known as the *critical point*) [20] for a given percentage α (< 1) in the standard normal distribution (e.g., 1.65 for 5 % and 2.33 for 1 %).

$$n = \frac{C + \sqrt{C^2 + \frac{8 \cdot C \cdot \sigma^2}{\theta^2}}}{2} \text{ if } n \geq 30, \text{ otherwise, } n = 30 \tag{11}$$

Here,

$$C = \{z(D)\}^2$$

If the input rate λ ($= 1/\theta$) is high (e.g., more than 100 tuples per second), which is common in real-world applications, we can simplify the above equation as follows.

$$n \approx \sqrt{2} \cdot z(D) \cdot \sigma\lambda \tag{12}$$

Note that our method based on the derived equation can provide worse performance than the existing method when we need to have highly accurate query results whose drop ratios are very small. Since $z(D)$ converges to an infinite value as D gets close to 0, the estimated buffer size can be impractically large when D is very small.

To resolve this problem, we utilize the existing adaptive method discussed in Section 2, when $D \leq 0.1\,\%$. Regarding this, we show through our experiments that the buffer size estimated by the adaptive method becomes smaller than that

of our method when $D \leq 0.1\,\%$. The results also show that the method keeps the percentage of tuple drops within $0.1\,\%$ when we set the parameter W (the interval of decreasing the max delay) to be large enough. Actually, we may not control the drop ratio in this method, nor can we guarantee that it always keeps the ratio within $0.1\,\%$. Nevertheless, these results show that the adaptive method can be a practical solution when we require high accuracy in query results.

3.3 Structure and Algorithm

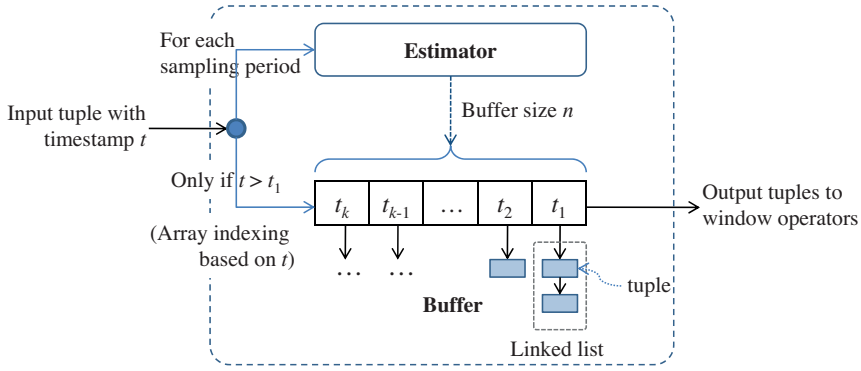


Fig. 3. Structure for processing out-of-order tuples

Figure 3 shows the structure to process out-of-order input tuples in our method, which consists of two main parts: the *buffer* and the *estimator*. The former maintains input tuples in an increasing order of their generation timestamps (hereafter, simply timestamps), and the latter estimates the size of the buffer base on Equation (11).

The buffer needs to be organized to accept input tuples efficiently while observing their timestamp order. The buffer size may significantly grow due to a high arrival rate of the input stream. Ordering tuples based on conventional sorting mechanisms providing $O(n \log n)$ time can be an exhausting job when the buffer size becomes huge.

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). Based on this assumption, we can easily group input tuples according to their timestamps by using an array whose elements correspond to tuples' timestamps. Actually, we organize the buffer as a circular array whose element has a list of tuples with same timestamps. Then, we map an input tuple to one of elements directly by array indexing based on the tuple's timestamp.

Whenever a new tuple arrives in the buffer, we first check its timestamp to determine whether we accept it or not. The determination is conducted based on the earliest timestamp of tuples in the buffer. If the tuple's timestamp t is smaller than the earliest timestamp t_1, the tuple is discarded. In other words, the earliest timestamp t_1 plays a role of the heartbeat in our method. If t is equal to t_1, we directly output the tuple to the window operator. Otherwise, we keep the tuple in the buffer. If the buffer becomes full by a new tuple, we send out an earliest tuple from the buffer whose timestamp is t_1. It is also possible to output more tuples with the same timestamps t_1 to make more space available in advance.

Algorithm 1. Periodical estimation of the buffer size

```
Input:  Tuple s with timestamp t
Data:   Counter c, Timestamp t_l
(1)   If c mod P_s = 0, then add s to the estimator buffer
(2)   If c mod P_e = 0
(3)       Estimate σ and θ from tuples in the estimator buffer
(4)       Estimate the buffer size n by using equation (11)
(5)       Keep the latest n tuples and send out other ones
(6)       t_l ← The earliest timestamp of tuples in the buffer
(7)   End if
(8)   If t < t_l
(9)           Discard s
(10)  Else if t = t_l
(11)      Send out s to a window operator
(12)  Else
(13)      Insert s to the buffer
          (The earliest tuple is sent out if the buffer is full)
(14)  End if
(15)  Increase c
```

Fig. 4. Algorithm for buffer size estimation in our method

The buffer size is controlled by the estimator periodically. We may estimate the buffer size for each tuple arrival. But, this approach may degrade overall performance due to heavy estimation cost. Instead, we estimate it for every P_e tuple arrivals. If the number of tuple arrivals per second (i.e., λ) is smaller than P_e, we conduct estimation every second. We also sample a tuple for every P_s arrivals from an input stream and keep at least M ($\geq P_e/P_s$) tuples in the buffer of the estimator to obtain the parameters σ and θ of Equation (11) (or Equation (12) if λ is large); to sample input tuples effectively, we may use more advanced techniques such as the *reservoir sampling* [21, 22]. Currently, P_e, P_s and M are predefined values in our method, and adaptive determination of the values (according to fluctuated stream characteristics) is left for future work.

Figure 4 shows an algorithm for estimating the buffer size periodically in our method. Whenever a new tuple s with timestamp t arrives, we first check the

sampling period by using the internal counter c and add the tuple to the sampling buffer of the estimator. Then, we check the estimation period also based on c. If so, we estimate the parameters σ and θ from tuples in the sampling buffer and calculate the buffer size by using Equation (11). After the buffer size n is identified, we maintain the latest n tuples in the buffer and send out all other tuples to window operators. We also refresh the heartbeat t_1 based on the remaining tuples in the buffer. Finally, we determine whether we accept or discard the given tuple in the buffer based on the earliest timestamp t_1 as discussed earlier.

Note that our algorithm in Figure 4 is used only when D is larger than 0.1 %. In other cases, we utilize the adaptive method as discussed in Section 3.2. When using the method, we keep the parameter W large enough to reduce as many tuple discards as possible; currently, we set W to be larger than or equal to λ. Our experimental results in the next section show that the adaptive method with such a large W keeps the percentage of tuple drops within 0.1 % in a typical situation.

4 EXPERIMENTAL RESULTS

In this section, we provide experimental results that compare our method with the adaptive method proposed by Srivastava et al. [12] whose estimation mechanism is briefly discussed in Section 2. We implemented a data generator to synthesize test data sets which satisfies our stream model discussed in Section 3.1. It receives parameters including a drop ratio D, an average interarrival rate of tuples λ, a mean of network delays μ and their standard deviation σ. To simplify our experiments, we currently set λ to 10 000 tuples per second and fixed the size of each data set to 1 000 000 tuples. Our experiments were conducted on Intel Pentium IV 2.4 MHz machine, running Window XP, with 1 G main memory.

We first tried to see the behavior of the adaptive method over streams whose characteristics are dynamically changed. To simulate the fluctuated stream, we change the parameters μ and σ periodically during the experiment. The parameters are randomly chosen among the values of "$0 \leq \mu \leq 6$" and "$0 \leq \sigma \leq 5$" in every 1, 3 or 5 seconds during the experiment. Then, we varied the parameter W and observed its influence to the buffer size and tuple drops. Figure 5 shows the progresses of the buffer size and the percentage of tuple drops according to changing W. As W increases, the buffer size becomes larger since W determines the period of decreasing the max delay m in the adaptive method; a large W will result in a large m as well as a large buffer. It is clear that the percentage of tuple drops gets reduced as the buffer size increases, which is shown in Figure 5 b).

Note that in this method, it is hard to determine the parameter W properly regarding to observing a user-specified drop ratio. Suppose that we need to keep the percentage of tuple drops within 1 %. Then, we may choose W between 100 and 1 000, but it is not clear which value satisfies the given drop ratio while providing a smaller buffer than other choices. Even though we obtain a proper value of W by trial-and-error, the chosen value may not satisfy the drop ratio in other streams

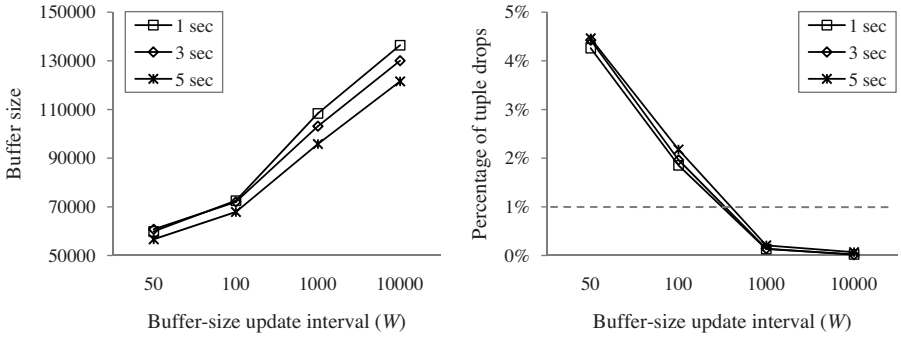


Fig. 5. Adaptive method: a) buffer size (left) and b) percentage of tuple drops (right)

which have different characteristics (e.g., "$0 \leq \sigma \leq 10$").

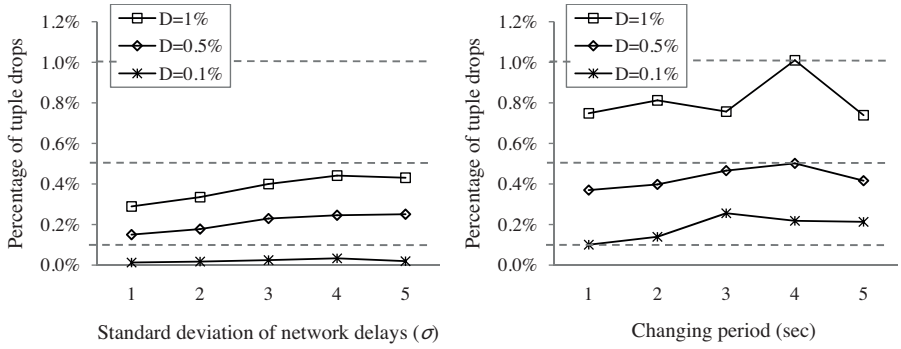


Fig. 6. Our method: percentages of tuple drops over a) a constant stream (left) and b) a changing stream (right)

On the other hand, our method enables a user to specify a drop ratio and adjusts the buffer size automatically to observe a given drop ratio. Figure 6 shows experimental results where our method observes a given drop ratio in most cases. The left figure shows the percentages of tuple drops when stream characteristics are not changed during the estimation. We set σ from 1 to 5, then observed tuple drops when the drop ratios are given to 1 %, 0.5 % and 0.1 % for each case. As we can see in the figure, there is no case violating a given drop ratio in this experiment.

When stream characteristics are dynamically changed, the drop ratio may not be strictly observed in the proposed method. In this experiment, we changed the stream characteristics in the same way of the previous one (i.e., "$0 \leq \mu \leq 6$" and "$0 \leq \sigma \leq 5$") and observed the percentages of tuple drops. Figure 6 b) shows its result for each case of the drop ratio given to 1 %, 0.5 % and 0.1 %. The violation occurs when we need to have highly accurate results as in the case of $D =$

0.1 %. The result leads to the necessity of more elaborate disorder control in our method.

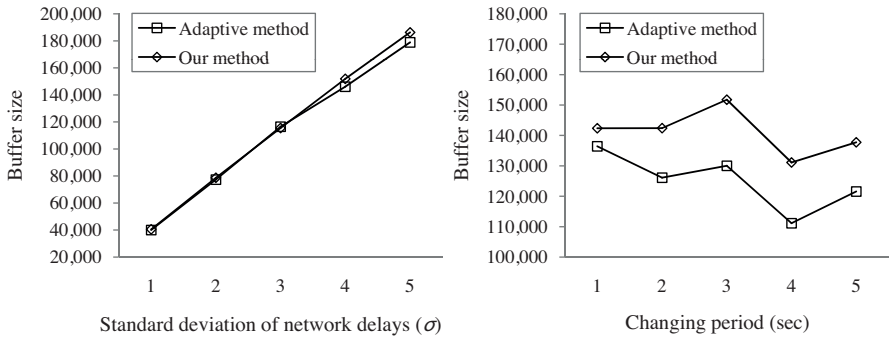


Fig. 7. Adaptive method vs. our method ($D = 0.1\,\%$): buffer sizes over a) a constant stream (left) and b) a changing stream (right)

As discussed earlier, our method can provide worse performance (i.e., a larger buffer) than the adaptive method when D is very small. To see this, we compared the buffer sizes estimated by our method (when $D = 0.1\,\%$) and the adaptive method. Figure 7 a) shows the buffer sizes of two methods according to increasing σ when stream characteristics are not changed. In this case, two methods provide similar performance. On the other hand, when stream characteristics are dynamically changed, the adaptive method shows better performance than our method, which is depicted in Figure 7 b); the performance gap between two methods increases up to 15 %.

The last two experimental results of Figures 6 and 7 show that more elaborate disorder control is required in our method. Note that, when stream characteristics are not changed, our method observes a given drop ratio and estimates the buffer size similar to that of the adaptive method. From this, we can see that our estimation function (i.e., Equation (11)) works properly. It seems that the estimation error and the larger buffer are due to the parameters μ and σ which are improperly estimated when stream characteristics are dynamically changed. Regarding this, we will have more study on this issue later (e.g., choosing proper sampling periods and sampling buffer sizes in Algorithm 1).

5 CONCLUSION AND FUTURE WORK

In this paper, we proposed a method to control disorder in continuous data streams. The proposed method enables a user to specify a drop ratio in a query, which denotes a percentage of tuple drops permissible in the disorder control process. By specifying the drop ratio, users can control the quality of query results according to application requirements. We derived our method based on a stream model where

tuples are randomly generated and arrive after normally-distributed network delays. The derived function utilizes a mean of tuples' interarrival times and a standard deviation of their network delays. This enables our method to control disorder adaptively in accordance with fluctuated stream characteristics. Our experimental results show that

- it is hard to control the amount of tuple drops in the existing method which adjusts the buffer size heuristically and
- the proposed method observes a user-specified drop ratio when it is larger than 0.5 %.

As shown in the experimental results, our method may violate a given drop ratio and provide a larger buffer than the existing method. This is when we need to have highly accurate results (whose drop ratio $D \leq 0.1\,\%$) over fluctuated streams. The results lead to the necessity of more elaborate disorder control in our method. Regarding this, we are planning to study how to choose a sampling period and a sampling buffer size properly over streams whose characteristics are dynamically changed.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2009-0089128).

Appendix: Distribution of tuple arrivals in our stream model

We will show that, if tuples are randomly generated with rate λ, their arrivals also become a Poisson process with rate λ, regardless of the distribution of network delays. As described in our stream model (Section 3.1), the i-th tuple S_i is generated from a remote source at time T_i and arrives at the destination at time U_i. Here, we assume that the network delay $(U_i - T_i)$ is an IID (Independent and Identically Distributed) random variable with distribution given by

$$F(x) \triangleq Pr\{U_i - T_i \leq x\} \tag{13}$$

with density

$$f(x) \triangleq \frac{\mathrm{d}F(x)}{\mathrm{d}x}. \tag{14}$$

Let us consider another stream model in a discrete time domain. We discretize time in small intervals δ. In this model, we assume that tuple arrivals on the intervals (i.e., discretized time slots) are independent and identical Bernoulli process with probability p.

Suppose that the i-th tuple $\hat{S}_i$ occurs at time slot $\hat{T}_i$ and arrives at time slot $\hat{U}_i$. The network delay $(\hat{U}_i - \hat{T}_i)$ can be represented as the number of slots between the

tuple occurrence and arrival, which is assumed to be an IID random variable with distribution given by

$$\hat{f}(x) \triangleq Pr\{\hat{U}_i - \hat{T}_i = x\}. \tag{15}$$

When δ converges to 0, two stream models based on a discrete and a continuous time domain become identical as long as

$$p = \lambda\delta \tag{16}$$

and

$$\hat{f}(x) = \delta \cdot f(x\delta). \tag{17}$$

To prove that the arrivals of tuples S_is are Poisson process is identical to showing that the arrivals of tuples $\hat{S}_i$s are Bernoulli. To prove it is Bernoulli, we have to prove the probability of a tuple arrival at each time slot of the destination is $\lambda\delta$ and it is memoryless.

The former one can be easily proven. The probability of a tuple arrival at a certain slot can be calculated as

$$\begin{aligned} & Pr\{\text{tuple arrival at } i^{\text{th}} \text{ slot}\} \\ &= \sum_j Pr\{\text{tuple occurs at } (i-j)^{\text{th}} \text{ slot}\} \cdot Pr\{\text{delivery takes } j \text{ slots}\} \\ &= \sum_j \lambda\delta \cdot \hat{F}(j) \\ &= \lambda\delta \sum_j \hat{F}(j) = \lambda\delta. \end{aligned}$$

The proof completes with the memoryless property. Let us assume that the arrival history during time q is known before a certain slot χ. q is finite time in a continuous time domain, but is able to have infinite number of slots with interval δ. However, there can be only finite number of arrivals during time q with probability 1 if the Poisson rate λ is finite. Thus, we assume that the history of arrivals during q is known and finite. Let Q be a set of slots with tuple arrivals during q. Now, we have the conditional probability of a tuple arrival on slot χ as

$$\begin{aligned} & Pr\{\text{tuple arrival on } \chi \text{ – arrivals on } Q \text{ during } q\} \\ &= \frac{Pr\{\text{arrival on } \chi\} \cdot Pr\{\text{arrival on } Q \text{ during } q \text{ — arrival on } \chi\}}{Pr\{\text{arrival on } Q \text{ during } q\}} \\ &= \frac{Pr\{\text{arrival on } \chi\} \cdot Pr\{\text{arrival on } Q \text{ during } q\}}{Pr\{\text{arrival on } Q \text{ during } q\}} \\ &= Pr\{\text{arrival on } \chi\}. \end{aligned}$$

From the above, tuple arrivals are memoryless. Based on the two results, we can see that tuple arrivals become a Poisson process if their generations have a Poisson distribution.

REFERENCES

[1] Abadi, D. et al.: Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal, Vol. 12, 2003, No. 2, pp. 120–139.

[2] Babcock, B.—Babu, S.—Datar, M.—Motwani, R.—Widom, J.: Models and Issues in Data Stream Systems. Proceedings of the ACM PODS 2002, Madison, Wisconsin, United States, 2002, pp. 1–16.

[3] Golab, L.—Ozsu, M.: Issues in Data Stream Management. ACM SIGMOD Record, Vol. 32, 2003, No. 2, pp. 5–14.

[4] Ghanem, T.—Hammad, M.—Mokbel, M.—Aref, W.—Elmagarmid, A.: Incremental Evaluation of Sliding-Window Queries over Data Streams. IEEE Transactions on Knowledge and Data Enginieering, Vol. 19, 2007, No. 1, pp. 57–72.

[5] Li, J.—Maier, D.—Tufte, K.—Papadimos V.—Tucker, P.: Semantics and Evaluation Techniques for Window Aggregates in Data Streams. Proceedings of the ACM SIGMOD 2005, Baltimore, Maryland, United States, June 2005, pp. 311–322.

[6] The STREAM Group: STREAM: The Stanford STREAM Data Manager. IEEE Data Engineering Bulletin, Vol. 26, 2003, No. 1.

[7] Tatbul, M.—Cetintemel, U.—Zdonik, S.—Cherniack, M.—Stonebreaker, M.: Load Shedding in a Data Stream Manager. Proceedings of the VLDB Conference 2003, Berlin, Germany, September 2003, pp. 309–320.

[8] Tatbul, M.—Zdonik, S.: Window-Aware Load Shedding for Aggregation Queries over Data Streams. Proceedings of the VLDB Conference 2006, Seoul, Korea, September 2006, pp .799–810.

[9] Chandrasekaran, S. et al.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. Proceedings of the CIDR 2003, Asilomar, California, United States, January 2003.

[10] Ghanem, T.—Aref, W.—Elmagarmid, A.: Exploiting Predicate-Window Semantics over Data Streams. ACM SIGMOD Record, Vol. 35, 2006, No. 1, pp. 3–8.

[11] Bose, S.—Fegaras, L.: Data Stream Management for Historical XML Data. Proceedings of the ACM SIGMOD 2004, Paris, France, June 2004, pp. 239–250.

[12] Srivastava, U.—Widom, J.: Flexible Time Management in Data Stream Systems. Proceedings of the ACM PODS 2004, Paris, France, June 2004, pp. 263–274.

[13] Maier, D.—Li, J.—Tucker, P.—Tufte, K.—Papadimos, V.: Semantics of Data Streams and Operators. Proceedings of the ICDT 2005, LNCS 3363, Edinburgh, Scotland, January 2005, pp. 37–52.

[14] Babu, S.—Srivastava, U.—Widom, J.: Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. ACM Transactions on Database Systems (TODS), Vol. 29, 2004, No. 3, pp. 545–580.

[15] Cranor, C.—Johnson, T.—Spataschek, O.—Shkapenyuk, V.: Gigascope: A Stream Database for Network Applications. Proceedings of the ACM SIGMOD 2003, San Diego, California, United States, June 2003, pp. 647–651.

[16] Johnson, T.—Muthukrishnan, S.—Shkapenyuk, V.—Spatscheck, O.: A Heartbeat Mechanism and Its Application in Gigascope. Proceedings of the VLDB Conference 2005, Trondheim, Norway, September 2005, pp. 1079–1088.

[17] Chen, J.—DeWitt, D.—Tian, F.—Wang, Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. Proceedings of the ACM SIGMOD 2000, Dallas, Texas, United States, May 2000, pp. 379–390.

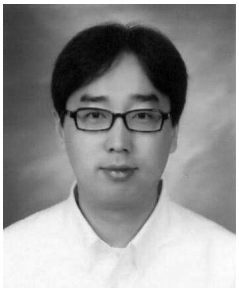
[18] Tucker, P.—Maier, D.—Sheard, T.—Fegaras, L.: Exploiting Punctuation Semantics in Continuous Data Streams. IEEE Transactions on Knowledge and Data Engineering, Vol. 15, 2003, No. 3, pp. 1–14.

[19] Ding, L.—Rundensteiner, E.: Evaluating Window Joins over Punctuated Streams. Proceedings of the ACM CIKM 2004, Washington, DC, United States, November 2004, pp. 98–107.

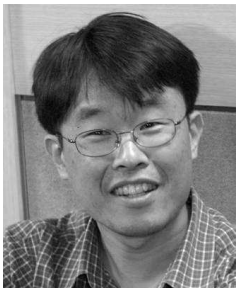
[20] Bertsekas, D.—Tsitsiklis, J.: Introduction to Probability: International Edition. Athena Scientific Press, Belmont, Massachusetts, 2002.

[21] Aggarwal, C.: On Biased Reservoir Sampling in the Presence of Stream Evolution. Proceedings of the VLDB Conference 2006, Seoul, Korea, September 2006, pp. 607–618.

[22] Babcock, B.—Datar, M.—Motwani, R.: Sampling from a Moving Window over Streaming Data. Proceedings of the ACM SIAM Symposium on Discrete Algorithms 2002, pp. 633–634.



Hyeon Gyu Kim received B. Sc. and M. Sc. degrees in Computer Science from University of Ulsan, Ulsan, Korea, in 1997 and 2000, respectively, and received his Ph. D. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2010. He has been a Senior Research Engineer at LG Electronics from 2001 to 2005, and a Senior Researcher at Korea Atomic Energy Research Institute (KAERI) since 2011. His research interests include data stream processing, mobile computing and probabilistic safety assessment.



Cheolgi Kim received a B. Sc. (1996) degree in computer science, followed by M. Sc. (1998) and Ph. D. degrees (2005) at Korea Advanced Institute of Science and Technology (KAIST). He has been working as a postdoc and a research scientist at University of Illinois since 2006. His research interests include safety-critical systems, cyber-physical systems, formal method applying for system integration, and mission-critical wireless networks.

Myoung Ho Kim received his B. Sc. and M. Sc. degrees in Computer Engineering from Seoul National University, Seoul, Korea in 1982 and 1984, respectively, and received his Ph. D. degree in Computer Science from Michigan State University, East Lansing, MI, in 1989. He joined the faculty of the Department of Computer Science at KAIST, Daejeon, Korea in 1989 where currently he is a Professor. His research interests include database systems, data stream processing, sensor networks, mobile computing, OLAP, XML, information retrieval, workflow and distributed processing.