Evaluation of Sliding Windows Based on Adaptive Disorder Control in Continuous Streams

Hyeon Gyu Kim[1] Cheolgi Kim[2] Myoung Ho Kim[1]

[1]Korea Advanced Institute of Science and Technology
Daejeon, South Korea

[2]Information and Communication Univerisity
Daejeon, South Korea

{hgkim,mhkim}@dbserver.kaist.ac.kr cheolgi@gmail.com

ABSTRACT

A sliding window is an important feature to process continuous streams. Various stream management systems evaluate sliding windows based on their estimation measures for disorder control. However, most of the measures do not reflect characteristics of input streams appropriately, and such lack of adaptivity may lead to inaccurate or delayed query results. To address this issue, we propose a method for evaluating sliding windows based on adaptive disorder control. We first present a structure of window operators using a 2-level index to handle tuples efficiently. Then, we propose an adaptive method base on the estimation measure that is derived from the distributions of tuple generation intervals and network latencies. The structure of window operators and estimation criteria for disorder control can be described declaratively in a query specification. This helps users to control the quality of query results such as accuracy or latency according to application requirements. Our experimental results show that the proposed measure provides better accuracy and stability than the one used in the existing method.

1. INTRODUCTION

There has been substantial research in the problem of processing continuous data streams based on the existing concepts of relational databases [1, 2, 6]. These efforts result in the advent of various *stream management systems*. The systems can be regarded as generalized platforms to process continuous queries over continuous streams in a real-time manner [7, 9]. Well-known examples of the systems are Stanford's STREAM [8], Berkeley's TelegraphCQ [15] and Aurora [13] / Borealis [14] from Brandeis, Brown, and M.I.T.

In stream management systems, a sliding window is an important feature of the continuous query [4, 5]. A window specifies a moving view that decomposes an unbounded stream into finite subsets called *window extents*. The window extents are commonly defined based on timestamps of tuples [12, 17]. When processing continuous streams, an extent can be viewed as a temporal relation. Based on the relation, blocking operators [7] such as sort or join, which are unable to start processing until an entire input is seen, can produce the results.

There are a few issues on the efficient evaluation of the sliding windows. First, since sliding windows are placed at the initial part of query operator trees, window evaluation operators are required to handle a huge amount of input tuples in a real-time manner. Second, to achieve semantic correctness, the window operators usually require that input tuples arrive in an increasing timestamp order. But when tuples are transmitted from remote sources, they may not arrive in the order they were sent due to various network latencies. Such out-of-order arrival of tuples complicates the process of determining the window extents. Disorder of tuples may have a considerable effect on the quality of query results such as accuracy or response time.

To resolve issues from disorder of tuples, the existing approaches usually employ fixed-size buffers or simple estimations based on the notion of *punctuations* [3]. The former case can be found in Aurora project [13]. In their research, they assume that a bound of network latency is known in advance, and from the assumption, the buffer has a fixed size that is large enough to cover the bound. The latter case is discussed in Jin Li's work [4] and *STREAM* project [8]. Briefly, a punctuation τ indicates that no more tuples having a timestamp greater than τ will be seen in the stream. Thus, when receiving τ, tuples having a timestamp less than or equal to τ can be processed in sliding windows. The punctuation can be either given by remote sources or estimated by a system.

However, there still remain some issues when applying the existing approaches to real-world stream applications. In buffer-based approaches, the bound of network latency may not be known in advance, and the bound is usually fluctuated. In that case, using a small-size buffer may cause too much tuples to be dropped, while using an excessively large-size buffer may result in high latency. In punctuation-based approaches, when the punctuations are estimated by a system, the estimation is usually conducted by *ad hoc* measures, not a theoretical one. In addition, the punctuation itself can be disordered by network latencies when given by remotes sources. Thus, it is also hard to expect accurate query results.

This paper presents a structure and method for evaluating sliding windows efficiently while resolving the stated issues by supporting adaptive disorder control. The proposed structure of window operators consists of the *record store* and the *disorder controller* as depicted in Figure 1. The record store maintains input tuples in an increasing timestamp order, and uses a 2-level index to place tuples efficiently. The disorder controller estimates the punctuations to decide a point of time to produce window extents from the tuples in the record store. When estimating the punctuations, the controller uses our measure that increases

adaptivity by reflecting characteristics of input streams appropriately. The proposed measure is theoretically derived from the distributions of tuple generation intervals and network latencies. In addition, if prior information about the network latency is available, the disorder controller can be configured to use a fixed-size buffer.

In our approach, the structure of window operators and estimation criteria for disorder control can be described declaratively in a query specification. This helps users to control the quality of query results according to application requirements. For example, as a parameter for the estimation, users can describe a *drop ratio* in the query specification, which denotes a percentage of tuple drops permissible in run-time processing. There is a trade-off between the drop ratio and the latency of the results, which will be described detail in the later section.

The rest of this paper is organized as follows. After a short review of related work in Section 2, Section 3 proposes a generalized structure of window operators using a 2-level index. Section 4 describes query language features for disorder control, and Section 5 explains derivation steps of the proposed measure for adaptive estimation. Section 6 discusses a method for evaluating sliding windows explicitly. Section 7 shows experimental results of our estimation measure and Section 8 concludes our discussion.

2. RELATED WORK

To resolve issues from disorder, two common approaches have been widely used. One is to maintain buffers to sort tuples, and the other is to leverage *punctuations* [3]. The former case can be found in *Aurora* [13]. The later case is discussed in Jin Li's work [4], *STREAM* [8] and *NiagaraCQ* [16].

In Aurora project [13], window operators simply ignore the out-of-order tuples. Instead, they use a buffer called *slack* to save the tuples by sorting them in the buffer. In their approach, they assume that a bound of network latency is always known in advance. From the assumption, the slack buffer has a fixed size of *n*. Any tuples that are more than *n* positions out of order are dropped. Note that, if a prior knowledge about the bound is not available and the bound is fluctuated, it may result in undesirable results.

In STREAM project [8], window operators do not have any mechanism for disorder control. Instead, the control is supported by external modules such as *Input Manager* [12] or prior operators that transmit the tuples to the window operators. In the Input Manager, the proposed solution for the control is based on the *heartbeats* [12], which can be thought of as special types of punctuations. When estimating the heartbeats, they use a simple measure that reflects the maximum difference of latencies [11, 12]. But such a measure may also result in inaccurate or delayed results from the lack of adaptivity.

Jin Li's work [4] proposes an efficient evaluation method of sliding windows for aggregate queries. And in their research, disorder control is conducted explicitly using the external punctuations given by remote sources. However, the external punctuations can be disordered by network latencies. In addition, their discussion of the evaluation method is only on aggregate queries. For example, a buffer in the window operator maintains only an intermediate result for aggregates, not an original tuples. Thus it cannot be used for other general types of queries. There is

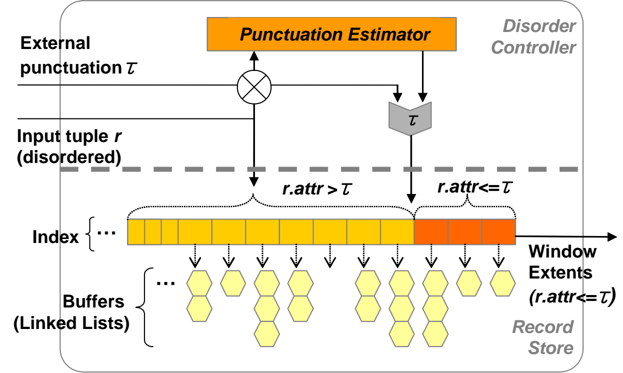

Figure 1. A structure of window operators

also an issue in efficiency if sliding windows are overlapped. If a window has a range of 5 minutes and is slid every 1 minute, the method requires 5 times of evaluation per every input tuples.

In NiagaraCQ [16], window operators leverage the external punctuations transmitted from the remote sources or internal punctuations estimated in the systems. However, if estimated internally, the punctuations are generated based on simple notions of the slack as in Aurora. Thus, there still remain same issues of upper approaches.

3. WINDOW OPERATORS

This section proposes a structure of window operators using a *2-level index* to handle tuples efficiently. The proposed structure consists of *record store* and *disorder controller* as depicted in Figure 1. The record store maintains a number of buffers to place the input tuples in an increasing order. And the disorder controller estimates the punctuations to decide a point of time to produce window extents from the tuples in the record store.

The record store uses an *index* to place input tuples into the buffers in a constant time. In our approach, the index is simply an array that maintains *pointers* to each *buffer* that has tuples of same timestamps, and an array index is directly mapped to a timestamp of the input tuples with a one-to-one correspondence. For example, a tuple having the timestamp *i* is placed to a buffer pointed by *index[i]*. From this configuration, the position of input tuples can be decided immediately. Furthermore, it doesn't need to sort tuples based on their timestamp, since the array indexes are already in an increasing order.

To illustrate the above process more detail, consider a query to inform the vehicles that are over-speeded in the latest 5 minutes from a highway. In this query, we suppose that each vehicle is equipped with a sensor for sensing its speed, and relays its sensor reading to a server every 30 seconds. And we assume that the sensor reading has a schema of <vehID, speed, ts>, which elements specify a vehicle ID, its speed, and the timestamp of the sensor reading each other. Based on these assumptions, the query can be described as Q1. In the query Q1, we use *CQL*-like language [10] with window syntaxes proposed in the Jin Li's work [4], where RANGE stands for the length of the window, SLIDE for the step by which the window moves, and WATTR for the windowing attribute – the attribute over which RANGE and SLIDE are specified.

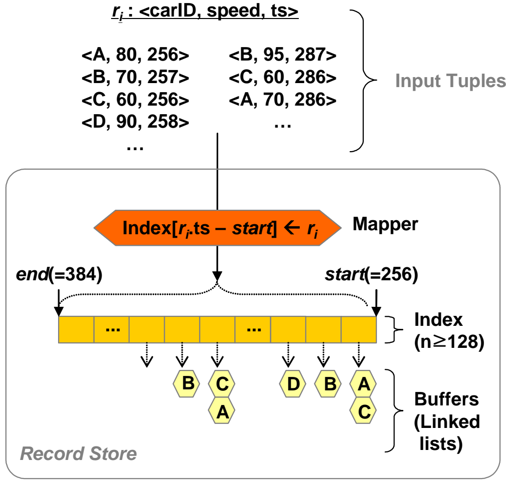

Figure 2. Placing tuples using an index

```
Q1: SELECT vehID, speed
    FROM Sensors [RANGE 300 seconds
                  SLIDE 30 seconds
                  WATTR ts]
    GROUP BY vehID
    HAVING AVG(speed) > 80
```

Figure 2 presents a detail scenario to show how to handle input tuples using the index for the above query. The upper part of the figure shows a number of input tuples, and the lower part presents the record store that consists of the *mapper*, the index and a number of the buffers. Whenever a tuple r_i arrives, the mapper calculates a difference between the timestamp of the tuple and a starting point of the index, which is denoted as *ts* and *start* in the figure. The difference determines an element of the index, which has a pointer to a buffer. The buffer maintains a number of tuples having the same timestamp and has a form of linked list. The linked list always keeps a track to the last node to insert a tuple without iteration.

The index has to be changed whenever it receives a tuple having a timestamp larger than the *end*, which means that the timestamp of the tuple exceeds the range covered by the index. In the example of Figure 2, the restructuring procedure is triggered when receiving a tuple having a timestamp larger than 384. The procedure incurs a number of array copies. Note that these copies may give a burden to the system if tuple arrivals are getting increased rapidly. In addition, the same situation can be occurred in case that the index is getting larger.

To resolve this problem, we propose a 2-level structure for the index. In the structure, the lower level index points each buffer and the upper level index indicates the lower level indexes. For convenience, we denote the lower level index as *p-node* and the upper level one as *i-node*. These two indexes are different in that the p-node always has a fixed size (n = 128), while the i-node has a variable size ($n \geq 8$). Thus, whenever restructured, the p-node is created or removed as an atomic unit, while the i-node is dynamically increased or decreased through array copies. However, the burden resulted from the array copies is negligible since the i-node has significantly small size compared with the p-node in our structure.

insertTuple (Tuple r_i) {
1. *i-index* ← (r_i.*attr* – *start*) / *sizeOf*(*P-Node*)
2. *p-index* ← (r_i.*attr* – *start*) % *sizeOf*(*P-Node*)
3. *p-node* ← *i-node*[*i-index*];
4. *p-node*[*p-index*] ← r_i

}

Figure 3. An algorithm for inserting tuples using the 2-level index

Figure 3 shows the algorithm for placing input tuples using the 2-level index. In the algorithm, the step 1 and 2 calculate the array indexes of i-node and p-node each other. The step 3 determines a p-node from the calculated index of i-node. And the step 4 inserts the given tuple into a buffer pointed by the p-node.

In our approach, a small size of our index can cover a relatively large size of sliding windows. For example, if an i-node has a size of 8, a p-node of 128 and a pointer of 4 bytes, then a total size of the index is 544 bytes. With this index, we can cover a sliding window having a range of 1024 epochs in the queries of sensor networks. The index can also handle about 18 minutes of sliding windows based on the timestamps.

4. QUERY SPECIFICATIONS

This section introduces query language features for disorder control. More specifically, the language features are used to specify the configuration of the disorder controller in our window operators. In the disorder controller, there are two ways of estimating punctuations using: 1) a *slack* buffer and 2) our measure. As noted earlier, the slack buffer has a fixed size and is used to sort input tuples in a bounded disorder. In case of using the slack buffer, a timestamp presented from the buffer can simply be regarded as a punctuation. When using our proposed measure, the punctuations are estimated in more sophisticated way. The details are described in the next section.

To describe the way of estimation in a query specification, we define two optional parameters: *SLACK* and *DRATIO*. The former denotes a size of the slack buffer and the latter a percentage of tuple drops permissible in run-time processing. These parameters are exclusive each other and can be defined in the window specification. If the former parameter is defined, the disorder controller conducts estimation using the slack buffer. In the other case, the controller estimates the punctuations based on our proposed measure.

As stated earlier, if a bound of network latencies is known in advance, the SLACK can be used for estimation. In this case, the value of the parameter is usually set to a large enough size to cover the bound. By specifying the size, users can get accurate query results without any tuple drops. The following shows how to define the SLACK in the specification. As language features for the window specification, we adopt syntaxes from the Jin Li's work [4].

```
Q2: SELECT vehID, speed
    FROM Sensors [RANGE 300 seconds
```

```
                    SLIDE 30 seconds
                    WATTR ts
                    SLACK 10]
```

If a prior knowledge of the network latencies is not available, it is more appropriate to use the DRATIO for estimation. Compared with the SLACK focusing only on accuracy, the DRATIO enables users to control the quality of query results according to their requirements. For example, if a small value of the drop ratio is given in the specification, users can get accurate query results with a small amount of tuple drops. Otherwise, users can obtain low latency in the results. The next example shows how to define the DRATIO.

```
Q3: SELECT vehID, speed
    FROM Sensors [RANGE 300 seconds
                  SLIDE 30 seconds
                  WATTR ts
                  DRATIO 1%]
```

In addition to define the parameters for estimating punctuations, we support another optional parameter called *BSIZE* to specify a limit on a total size of the buffers in the record store. The parameter can be declared with either SLACK or DRATIO as follows.

```
Q4: SELECT vehID, speed
    FROM Sensors [RANGE 300 seconds
                  SLIDE 30 seconds
                  WATTR ts
                  DRATIO 1%
                  BSIZE 100]
```

If the BSIZE is used together with the one of SLACK or DRATIO, it has a higher priority than the one in run-time estimation. That is, in upper example, the disorder controller starts estimation based on the given DRATIO of 1%. But if the total size of the buffers exceeds the given BSIZE of 100, the size is fixed to 100. In this case, the DRATIO has no effect on estimation until the size returns to be smaller than 100.

5. ADAPTIVE ESTIMATION

This section describes derivation steps of our measure for estimating punctuations. The section starts with preliminaries such as problem statements and some assumptions, and then explains derivation steps of the measure based on the assumptions. At the end of this section, we give an algorithm for estimating punctuations and discuss time and space complexities.

5.1. Preliminaries

In our approach, tuple drops are controlled by a drop ratio which is defined in a query specification. A disorder controller should carefully estimate punctuations to keep a total number of tuple drops from violating the given drop ratio. Note that a tuple drop is presented whenever the tuple carries a timestamp less than or equal to a punctuation previously estimated.

Let τ_p be an application timestamp of the punctuation to be estimated and $\mathcal{T}_{n+1}$ be a random variable for an application timestamp of tuple that will be arrived after the punctuation τ_p. Then an expression to estimate the punctuation can be written as follows.

$$\{ \tau_p \in \max(\mathcal{T}) \mid \Pr(\mathcal{T}_{n+1} < \mathcal{T}) < \Pr_d \text{ for some } \mathcal{T}, \mathcal{T}_{n+1} \in \mathrm{T} \} \quad \ldots (5.1)$$

For convenience, in the remaining part of this paper, we use conventions that $\mathcal{T}_i$ denotes a random variable for an application timestamp of a tuple and T_i a variable for a system timestamp of the tuple. All of the timestamps is assumed to be elements of a discrete and ordered time domain T. In addition, $\Pr_d$ denotes a drop ratio of the parameter DRATIO given in a specification.

In order to derive an estimation measure, we make two assumptions such that an interval of tuple generations in stream sources has an exponential distribution with a mean of θ (5.2), and a transmission delay from different network latencies follows a normal distribution with a mean of μ and a standard deviation of σ (5.3).

$$(\mathcal{T}_i - \mathcal{T}_{i-1}) \sim \mathrm{Exp}(\theta) \quad \ldots (5.2)$$

$$(\mathrm{T}_i - \mathcal{T}_i) \sim \mathrm{N}(\mu, \sigma) \quad \ldots (5.3)$$

In above assumptions, the θ, μ and σ can be deducted by sensing a number of latest tuples. For this purpose, we introduce a circular list called *VSeq*, which accumulates timestamp information of the latest tuples arrived at a disorder controller. The size of VSeq is continuously changed according to estimation results for the upper distributions, and in our approach, it is always larger than or equal to 30.

Based on the information of VSeq, the θ can be calculated simply by the following equation (5.4), where n specifies the size of VSeq, T_1 a system timestamp of the earliest tuple in VSeq, and T_n a system timestamp of the latest one.

$$\theta = (T_n - T_1) / n \quad \ldots (5.4)$$

The μ and σ can also be estimated from VSeq. The following steps show how μ is estimated. It simply removes the earliest delay and adds a new delay to get the sum of delays in VSeq, and then divide the sum with n to get μ. The steps for estimating σ are similar with these. In the below, *tail* and *head* denotes each pointer indicating a tail node and a head node in the VSeq, and r is the latest tuple arrived.

$sum \leftarrow sum - (VSeq[tail].T - VSeq[tail].\mathcal{T}) + (r.T - r.\mathcal{T})$;
$\mu \leftarrow sum / n$;
$VSeq[head].T \leftarrow r.T$;
$VSeq[head].\mathcal{T} \leftarrow r.\mathcal{T}$;

5.2. Derivation of Our Measure

This part of the section explains derivation steps of our measure for estimating punctuations based on above assumptions. Before discussing the steps, we present the distribution of an interval between the earliest system timestamp and the latest one. In the below expression, T_1 denotes the earliest one and T_n the latest one.

$$(T_n - T_1) \sim N((n-1)\cdot\theta, \sqrt{(n-1)\cdot\theta^2}) \quad \ldots (5.5)$$

Derivation:

Let $\mathcal{V}_i$ be an interval $(T_i - T_{i-1})$ between system timestamps of consequent tuples in VSeq, then $\mathcal{V}_i$ also has an exponential distribution from the assumption (5.2). The mean θ of the distribution can also be calculated by the equation (5.4).

Since number of $\mathcal{V}_i$ is equal to n-1 and the n is larger than or equal to 30 in our approach, it is large enough to approximate

the sum of $\mathcal{V}_i$ to a normal distribution according to the *Central Limit Theorem* [18]. In the distribution of the sum of $\mathcal{V}_i$, that is $(T_n - T_1)$, a mean of $(T_n - T_1)$ can be calculated as a mean of $\mathcal{V}_i$ multiplied by n-1 because of the independence of $\mathcal{V}_i$, and a standard deviation of $(T_n - T_1)$ is obtained in a same way, where the mean and the standard deviation of $\mathcal{V}_i$ is θ and θ^2 each other since $\mathcal{V}_i$ follows an exponential distribution. □

Based on the distribution (5.5) and the previous assumptions, it can be predicted whether a future tuple is dropped or not. When predicting such a drop of the next tuple, we assume that the generation interval and arrival of the tuple follow the current distributions. That is, the future tuple is generated after θ from the time that the last tuple is occurred and transferred to a system after μ with the variance of σ, where the θ, μ and σ are the parameters currently estimated.

The following equation is to estimate a drop ratio of the future tuple based on the information of VSeq. In the below, $\mathcal{T}_{n+1}$ is a random variable for an application timestamp of the tuple and $\mathcal{T}_1$ a variable for the earliest timestamp in VSeq, which is same as a current punctuation $\mathcal{T}_p$.

$$\Pr(\mathcal{T}_{n+1} < \mathcal{T}_1) = Z\left(\frac{n\theta}{\sqrt{2\sigma^2 + n\theta^2}}\right) \quad \ldots (5.6)$$

Derivation:

From the upper equation, the left term can be rewritten as follows.

$$\Pr(\mathcal{T}_{n+1} < \mathcal{T}_1) = \Pr(\mathcal{T}_{n+1} - \mathcal{T}_1 < 0)$$

Again, $\Pr(\mathcal{T}_{n+1} - \mathcal{T}_1 < 0)$ can be decomposed with three sub terms as shown in Figure 4. In the following, T_i is a system timestamp corresponding to an application timestamp $\mathcal{T}_i$.

$$(\mathcal{T}_{n+1} - \mathcal{T}_1) = (\mathcal{T}_{n+1} - T_{n+1}) + (T_{n+1} - T_1) + (T_1 - \mathcal{T}_1)$$

In the decomposition, both random variables relevant to the sub terms $(\mathcal{T}_{n+1} - T_{n+1})$ and $(T_1 - \mathcal{T}_1)$ follow a normal distribution from the assumption (5.3).

$$(\mathcal{T}_{n+1} - T_{n+1}) \sim \mathrm{N}(-\mu, \sigma) \quad \ldots (5.6.1)$$

$$(T_1 - \mathcal{T}_1) \sim \mathrm{N}(\mu, \sigma) \quad \ldots (5.6.2)$$

Also a random variable relevant to the sub term $(T_{n+1} - T_1)$ has a normal distribution from the derived distribution (5.5).

$$(T_{n+1} - T_1) \sim \mathrm{N}(n\cdot\theta, n\cdot\theta^2) \quad \ldots (5.6.3)$$

Using the derived distributions from (5.6.1) to (5.6.3) and *MGF* (*Moment Generating Function*) of a normal distribution [18], The term $(\mathcal{T}_{n+1} - \mathcal{T}_1)$ can be transformed as follows.

$$\begin{aligned} M_{\mathcal{T}_{n+1}-\mathcal{T}_1}(s) &= M_{\mathcal{T}_{n+1}-T_{n+1}}(s)\cdot M_{T_{n+1}-T_1}(s)\cdot M_{T_1-\mathcal{T}_1}(s) \\ &= e^{(\delta^2 s^2/2)-\mu s}\cdot e^{(n\theta^2 s^2/2)+n\theta s}\cdot e^{(\delta^2 s^2/2)+\mu s} \\ &= e^{\{(\delta^2 s^2/2)-\mu s\}+\{(n\theta^2 s^2/2)+n\theta s\}+\{(\delta^2 s^2/2)+\mu s\}} \\ &= e^{\{(2\delta^2+n\theta^2)s^2/2+n\theta s\}} \end{aligned}$$

The result is again in the form of normal distribution MGF. From this, a random variable $(\mathcal{T}_{n+1} - \mathcal{T}_1)$ follows a normal distribution such as:

$$(\mathcal{T}_{n+1} - \mathcal{T}_1) \sim N(n\cdot\theta, \sqrt{2\sigma^2 + n\cdot\theta^2})$$

After normalization of the above, we can finally obtain an equation for estimating a probability of the future tuple drop.

$$\Pr(\mathcal{T}_{n+1} < \mathcal{T}_1) = Z\left(\frac{n\theta}{\sqrt{2\sigma^2 + n\theta^2}}\right) \; □$$

Given VSeq, the equation (5.6) estimates a probability of the future tuple drop. Observe that if we have a $\Pr_d$ from a query specification, it is possible to obtain a size of VSeq after slight modification of the equation (5.6). An equation to get the size has a form of the following: In the right term of equation (5.6), n becomes a variable intended to be estimated, while a given $\Pr_d$ becomes a constant instead of the left term of the equation.

The next expression is to get the size of VSeq when $\Pr_d$ is given. In the below, c is a constant which is a square of z-value corresponding to the given $\Pr_d$, and N denotes a set of natural numbers.

$$\{ n_p \in \max(n) \mid n^2 - c\cdot n - 2\cdot c\cdot\sigma^2 / \theta^2 < 0 \text{ for some } n \in \mathrm{N} \} \quad \ldots (5.7)$$

Derivation:

As stated earlier, this is a form of an inequality that n becomes a variable intended to be estimated in the equation (5.6). The n_p denotes the maximum value of n which satisfies the above

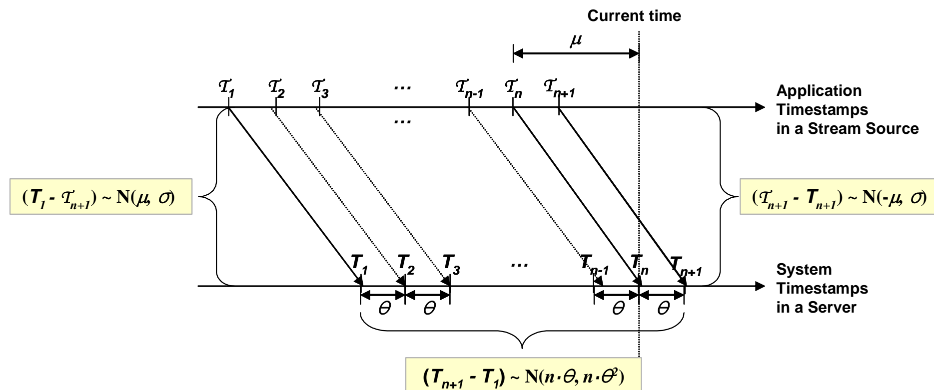

Figure 4. Decomposition of $(\mathcal{T}_{n+1} - \mathcal{T}_1)$

expression.

$$Z\left(\frac{n\theta}{\sqrt{2\sigma^2 + n\theta^2}}\right) < Z(\mathrm{Pr_d}) \rightarrow$$

$$\frac{n\theta}{\sqrt{2\sigma^2 + n\theta^2}} < Z(\mathrm{Pr_d}) \rightarrow$$

$$(n\theta)^2 < Z(\mathrm{Pr_d})^2 (2\sigma^2 + n\theta^2) \rightarrow \ldots$$

The desired result can be obtained by derivations from the above left-most inequality toward a right direction. □

From the size of VSeq satisfying the given Pr_d, a punctuation τ_p can be easily obtained. The equation for this purpose is described in the following. In the below equation, t_n denotes the latest system timestamp in VSeq.

$$\tau_p = \tau_n - n_p \cdot \theta \;\; (\tau_n = t_n - \mu) \qquad \ldots (5.8)$$

Derivation:

Remind that an interval between all consequtive tuples in VSeq is θ as discussed in earlier part. From this, given n_p, a region satisfying the current Pr_d can be calculated by $n_p \cdot \theta$. In addition, a maximum application timestamp τ_n can simply be obtained by (t_n - μ) from the assumption of network delay (5.3). Consequently, the punctuation τ_p satisfying the Pr_d is derived by subtracting the region $n_p \cdot \theta$ from the maximum application timestamp τ_n. □

5.3. Algorithm

In the previous part of this section, we explain derivation steps to estimate punctuations based on the information maintained in VSeq. In the derivation steps, the size of VSeq satisfying the given Pr_d is calculated by the inequality (5.7), and from the size n_p, the punctuation τ_p is simply obtained by the equation (5.8). These two steps are continuously performed by a disorder controller whenever a new tuple arrives.

The algorithm described in Figure 5 shows these steps written in pseudo codes. In the algorithm, *Eqn7* denotes a function playing a role of the inequality (5.7) and *Eqn8* a function of the equation (5.8). In addition we denote a VSeq having a size of n as VSeq(n).

Whenever a new tuple comes in, timestamp information of the tuple is accumulated in VSeq first, and then the function *Ready* checks whether VSeq is filled up. If not so, an estimation process is not activated and a previously calculated punctuation is simply returned. These are described in steps 1 and 2.

If the VSeq is ready, the estimation process is performed using a sequence of steps 3 to 6. The first step 3 is to estimate parameters such as μ, σ and θ, which can be easily derived from VSeq in a constant time, and already discussed in the first part of this section. The step 4 calculates the size of VSeq satisfying the given Pr_d using the inequality (5.7). To get the size n_p in a constant time, we substitute the inequality with an equation mark and just apply a floor function for a calculated n_p. After this, the punctuation τ_p is obtained by the equation (5.8) using the n_p, which is described in the step 5. The step also takes a constant time apparently.

The step 6 resizes the VSeq according to n_p calculated from the step 4. Unfortunately, this step may have a time complexity of

```
Adaptive (tuple r)
1. VSeq(n_p) ← t and τ of r;
2. If (Ready(VSeq)==true) {
3.     μ, σ and θ ← VSeq(n_p);
4.     n_p ← Eqn7(c, σ, θ);
5.     τ_p ← Eqn8(t, n_p, θ);
6.     VSeq ← VSeq(n_p);
7.}
8. return τ_p;
```

Figure 5. An algorithm for estimating punctuations using our adaptive measure

O(n) in some cases. If the estimated size of n_p is larger than or equal to the current size, there is only a processing effort to add new nodes in the circular list. Furthermore, no estimation will be occurred until the list is filled up. From these reasons, we can say that it takes a constant time in this case. However, in a converse case, a number of nodes should be removed one by one from the list and a sum of the timestamps also be refreshed. Such a case requires an iteration, which means a time complexity of O(n), and may give a burden to a system when overloaded situations.

From the above perspective, the time complexity of our algorithm is O(n) for any tuple arrival. In addition, since our algorithm only uses VSeq having a size of n_p when estimating punctuations, a space complexity is basically O(n_p). If a buffer is used to present out-of-order tuples in an increasing order, the required space is increased by an amount of the buffer size n_s, thus the space complexity is O($n_p + n_s$) in this case.

6. WINDOW EVALUATION

This section discusses a method for evaluating sliding windows explicitly. More specifically, it is about decision of which window extents can be produced from the tuples maintained in the record store when given the estimated punctuations. In the section, we first present a condition for explicit processing, and then discuss some issues required for the processing.

Generally, a window extent can be presented from the window operator when the minimum timestamp of the extent is larger than the given punctuation. For example, given RANGE of n and SLIDE of s, i-th window extent can be described as (6.1), and it can be presented when the condition (6.2) is satisfied with the given punctuation τ_p. In the below, R is a set of tuples, r a tuple in the R and ts a timestamp of the tuple.

$$\textbf{\textit{extent(i)}} = \{ r \in \mathrm{R} \mid (i+1)*s - \;\; n \le r.ts < (i+1)*s \} \quad \ldots (6.1)$$

$$\tau_p \le (i+1)*s - \;\; n \qquad \ldots (6.2)$$

In order to process window extents consistently based on the condition (6.2), the punctuations has to be monotonously increased. Otherwise, same extents can be produced multiple times or dangling tuples, which are not included in any extents, are occurred, since window operators usually ignore out-of-ordered tuples having smaller timestamps than the "latest" punctuation. This issue can be easily resolved by sending the

punctuations in a monotonic way. That is, if the latest punctuation is smaller than the previous one, the disorder controller just sends the previous one.

In our approach, window extents can be presented only if the condition (6.2) is satisfied. It is independent of the tuple arrivals. If there is no disorder, it is natural to suppose that each extent can be produced regularly based on the tuple arrivals or sliding intervals. However, if input tuples are out-of-ordered, it is hard to produce the extents regularly because some latency is required to accumulate the out-of-ordered tuples and sort them.

Our window operators insert a special type of tuple called *sync tuple* to the end of every window extents. A sync tuple denotes the boundary of each window extent, so that other operators can recognize each extent with it. The sync tuple can also be used to produce meaningful results when joining multiple streams. To illustrate the usage, suppose a query to decide whether to pay a toll to vehicles in a highway according to its traffic condition. Assume that the highway is congested if an average speed is less than 40 mph, then vehicles in the highway pay a toll. Let each vehicle have a sensor that relays its speed information every 30 seconds. Figure 6 shows a simplified plan tree for this query, where each oval denotes an operator in the tree. In the figure, OP_SRC denotes an operator to store input tuples, OP_WIN a sliding window operator, OP_AVG an aggregate operator calculating averages, and OP_JOIN a join operator.

In Figure 6, an upper part of the query tree is to get active vehicles in the highway by maintaining tuples of the latest 30 seconds, and the lower part checks the traffic condition of the highway. Thus, by joining these two streams, we can decide whether to pay a toll to vehicles. However, if the two streams are not synchronized, that means only one of the two streams is continuously processed, undesirable situations can happen. For example, the query can pay a toll to vehicles although the traffic condition is not congested.

7. EXPERIMENTAL RESULTS

This section presents experimental results of our estimation measure in terms of accuracy and stability. For this purpose, we compared our measure with the existing approach where the measure reflects the maximum difference of latencies for estimating punctuations [12]. We conducted experiments in two ways: 1) The first experiment compares drop ratios of each measure in terms of accuracy, and 2) The second experiment observes how significantly each measure is affected by exceptional cases in terms of stability.

In order to conduct experiments, we implemented a window operator as proposed in the paper and connected it to TinyDB [19] for data generation. We varied the configuration of TinyDB to get data from 16 to 20 sensors in every second, and collected a number of data sets for each case. Among the data sets, we finally selected 20 data sets in which network latencies follow a normal distribution, since our estimation measure is derived based on such an assumption as stated in Section 5.

	Existing Approach			Our Approach (DRATIO=1%)		
Data Set	Buffer size	Latency	Drop ratio	Buffer size	Latency	Drop ratio
~		~			~	
8	97	6.15	0.91	67	4.16	0.73
9	31	1.85	1.12	33	1.87	0.00
10	50	3.18	1.03	58	3.57	0.09
11	40	2.58	0.85	50	3.26	0.00
12	46	2.80	1.24	56	3.25	0.00
~		~			~	
Results	**65.30**	**4.22**	**1.10**	**68.75**	**4.33**	**0.51**

Figure 7. Experimental results from the existing measure and our measure

Figure 7 compares experimental results from both measures in terms of three parameters: buffer size, latency and drop ratio. The first denotes an average size of the buffers in the window operator, and the second is an average waiting time of tuples in the buffers. The third means a resulting ratio of tuple drops per total number of input tuples. In the bottom line of the figure, average values of each column are presented.

As our experimental results, our measure shows a resulting drop ratio of 0.51% when the DRATIO is given to 1% in a query specification, while the existing measure shows a ratio of 1.10%. In terms of buffer sizes and the latencies, the existing measure shows a smaller size than our measure, but the differences are not significant.

One notable thing from the results is that it is hard to use the existing measure in applications that require strict criteria about accuracy of query results. For example, consider an application that only allows query results having tuple drops less than 1%. In case of the existing measure, we observed 13 times of violation that exceeds a drop ratio of 1% during the experiments. Moreover, in the measure, there is no way to control the tuple drops that makes the ratio be lower.

In order to test stability of both measures, we placed a number of tuples having a large size of disorder in the early stage of the data sets, and increased the size to observe relationships with the

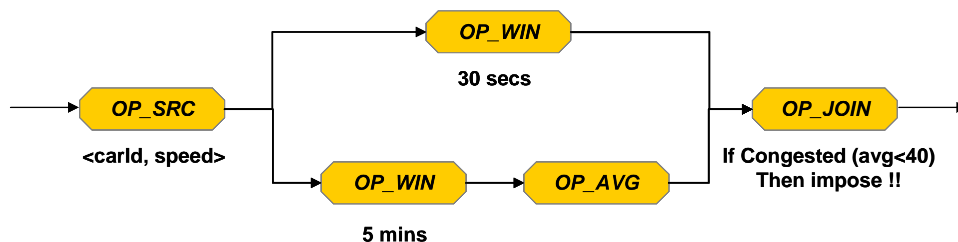

Figure 6. A simplified query tree for paying a toll according to traffic conditions

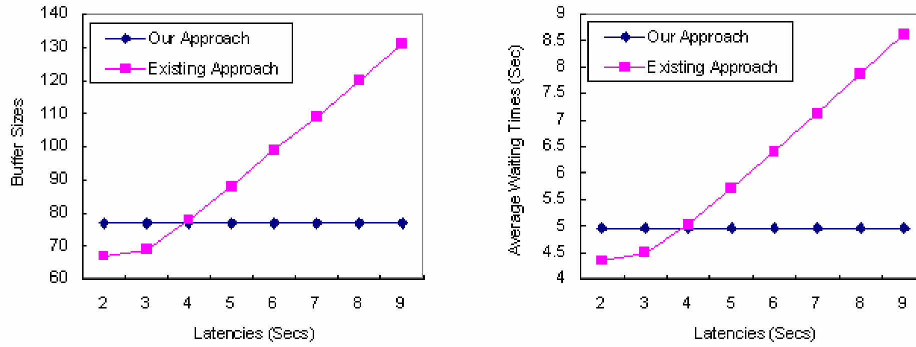

Figure 8: Relationship between disorder sizes of an early stage and (a) buffer sizes and (b) average waiting times

buffer sizes or the average waiting times. Figure 8 shows results from the experiments. As shown in the figure, the existing measure has a close relationship with the disorder sizes of the early stage, while our measure is not affected by the exceptional cases.

There have been also proposed an estimation measure to use an average value of the maximum disorder to resolve problems from the lack of adaptivity in the existing approaches [12]. However, we observed that the resulting drop ratios are increased to more than 3% in the measure, while it is not affected by the exceptional cases so much. Therefore, it is also hard to use such a measure in the applications that require strict accuracy.

8. CONCLUSION

This paper presents a structure and method for evaluating sliding windows efficiently. The proposed structure consists of the record store and the disorder controller. The record store maintains input tuples in an increasing order and uses a 2-level index to place the tuples in a constant time. The disorder controller estimates punctuations to decide a point of time to produce window extents from the tuples in the record store. The window structure and its behavior can be described in a query specification using an SQL-like language. For this purpose, we have introduced a few optional parameters such as SLACK, DRATIO and BSIZE. Based on the parameters given in the specification, run-time estimation is conducted using our proposed measure which is derived theoretically from the distributions of tuple generation intervals and network latencies. To verify adaptivity of our method based on the proposed measure, we have compared it with an existing method in terms of accuracy and stability and have shown that our method works better than the one proposed earlier.

Our estimation measure can be extended to cover various causes of disorder in data streams such as merging unsynchronized streams or data prioritization. We are planning to address these issues to make our method more scalable and flexible.

9. REFERENCES

[1] Douglas Terry, David Goldberg, David Nichols, and Brian Oki, *Continuous Queries over Append-Only Databases*. ACM SIGMOD, 1992.

[2] Samuel R. Madden, Mehul A. Shah, Joseph M. Hellerstein and Vijayshankar Raman, *Continuously Adaptive Continuous Queries over Streams*. ACM SIGMOD Conference, Madison, WI, June 2002

[3] Peter A. Tucker, David Maier, Time Sheard, Leonidas Fegaras, *Exploiting Punctuation Semantics in Continuous Data Streams*. IEEE Transactions on Knowledge and Data Engineering, May/June 2003.

[4] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter A. Tucker, *Semantics and Evaluation Techniques for Window Aggregates in Data Streams*. ACM SIGMOD 2005,

[5] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, Peter A. Tucker, *No Pane, No Gain: Efficient Evaluation of Sliding Window Aggregates over Data Streams*. SIGMOD Record, Vol 34, No. 1, March 2005.

[6] S. Babu and J. Widom, *Continuous Queries over Data Streams*. ACM SIGMOD Record, Sep. 2001.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, *Models and Issues in Data Stream Systems*. Invited paper in Proc. of the 2002 ACM Symp. on Principles of Database Systems (PODS 2002), June 2002.

[8] Arvind Arasu et al, *STREAM: The Stanford Data Stream Management System*. IEEE Data Engineering Bulletin, Vol. 26 No. 1, March 2003.

[9] Rajeev Motwani et al, *Query Proessing, Resource Management, and Approximation in a Data Stream Management System*. CIDR 2003, Jan. 2003.

[10] A. Arasu, S. Babu and J. Widom, *The CQL Continuous Query Language: Semantic Foundations and Query Execution*. Stanford University Technical Report, Oct. 2003.

[11] S. Babu, U. Srivastava and J. Widom, *Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams*. ACM TODS, Sep. 2004.

[12] U. Srivastava and J. Widom. *Flexible Time Management in Data Stream Systems*. ACM PODS 2004, June 2004.

[13] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik. *Aurora: A New Model and Architecture for Data Stream Management*. VLDB Journal (12)2: 120-139, August 2003.

[14] D. Abadi at al, *The Design of the Borealis Stream Processing Engine*. CIDR 2005, Asilomar, CA, January 2005.

[15] Sirish Chandrasekaran et al, *TelegraphCQ: Continuous Dataflow Processing for an Uncertain World*. CIDR 2003.

[16] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. *NiagaraCQ: A scalable continuous query system for internet databases*. ACM SIGMOD pages 379–390, May 2000.

[17] Sujoe Bose and Leonidas Fegaras, Data Stream Management for Historical XML Data, ACM SIGMOD, June 2004.

[18] Dimitry P. Bertsekas and John N. Tsitsiklis, *Introduction to Probability: International Edition*, Athena Scientific, Belmont, Massachusetts, 2002.

[19] TinyDB: http://www.tinyos.net.