A Review of Window Query Processing for Data Streams

Hyeon Gyu Kim*
Division of Computer, Sahmyook University, Seoul, Korea
hgkim@syu.ac.kr

Myoung Ho Kim
Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea
mhkim@dbserver.kaist.ac.kr

Abstract
In recent years, progress in hardware technology has resulted in the possibility of monitoring many events in real time. The volume of incoming data may be so large, that monitoring all individual data might be intractable. Revisiting any particular record can also be impossible in this environment. Therefore, many database schemes, such as aggregation, join, frequent pattern mining, and indexing, become more challenging in this context. This paper surveys the previous efforts to resolve these issues in processing data streams. The emphasis is on specifying and processing sliding window queries, which are supported in many stream processing engines. We also review the related work on stream query processing, including synopsis structures, plan sharing, operator scheduling, load shedding, and disorder control.

Category: Ubiquitous computing

Keywords: Data streams; Continuous queries; Sliding windows; Window query processing; Load shedding

I. INTRODUCTION
The past few years have witnessed the emergence of applications that monitor streams of data items, such as sensor readings, network measurements, stock exchanges, online auctions, and telecommunication call logs [1-4]. In these applications, fast, approximated results are more meaningful than delayed, accurate results. For example, consider a medical center where biosensors are used to monitor the body status of patients. In this example, a life-threatening event should be detected on time, and notified to the medical staff immediately, even if it proves to be a false alarm. Delayed detection of critical events is unacceptable. Similar examples can be found in network intrusion detection, plant monitoring, and so on.

Stream monitoring applications do not fit the traditional database model. Data streams are potentially unbounded. Therefore, it is not feasible to store an entire stream in a local database. Also, queries posed to a database system may not be answered, because an input stream can be infinite. Instead, stream applications adopt the notion of continuous queries. In this scheme, queries over data streams run continuously over a period of time and incrementally return new results, as new data arrives.

For example, consider a query that asks for the maximum value of sensor readings over the latest 30 seconds. This can be specified as a structured query language (SQL)-like query $Q1$, where window specification is defined in square brackets.
Q1. SELECT MAX(value)
FROM Sensors [RANGE 30 seconds]

During run time, the query is evaluated for each tuple arrival. Whenever a new tuple arrives at time $t$, windowing tuples is first performed: tuples whose timestamps are within $(t - 30, t]$ belong to the current window, and other tuples are discarded. Then, the maximum sensing value over tuples in the window is calculated. The same process is repeated, whenever a tuple arrives on the input stream.

As shown in the example, continuous queries generally use sliding windows, to limit the scope of query processing to recent data [3-6]. From this, the queries can be answered over unbounded data streams, even when they involve blocking operators, such as joins and aggregates—operators that cannot start processing until entire inputs are ready.

Fig. 1 shows a system structure to process window queries. The query processing engine is also called a data stream management system (DSMS). Well-known examples of the system include Aurora [1], STREAM [4], TelegraphCQ [7], Gigascope [8], NiagaraCQ [9], and StreamMill [10]. The system can be viewed as a real-time processing engine with the support of an SQL interface, which enables users to specify their applications in a declarative fashion. It consists of the following components:

- **Query compiler** translates user-specified window queries into a query plan tree. The plan tree can be viewed as a filter, through which input streams pass. As mentioned above, queries specified for stream applications can be characterized as window queries (Section II).
- **Query manager** runs the generated plan trees over input data streams. Some parts of the trees can be shared to reduce execution time (Section III-A). The behavior of stateful operators, such as aggregates and joins, can differ from those in the traditional DBMSs (Section III-B). The operators in the plan trees can be scheduled dynamically, according to changing system status (Section III-C).
- **Memory manager** provides storage for input tuples and intermediate results generated from the operators in plan trees. It is organized to maximize the sharing of tuples, to avoid disk accesses by reducing memory size (Section III-A).
- **Load shedder** monitors system status including arrival rates of input tuples and memory growth during query execution. If the system is overloaded, it prompts the memory manager or the input manager to discard some portion of the tuples (Section III-D).
- **Input manager** synchronizes multiple data streams (for a join) and resolves disorder of the streams. Disorder control is necessary to clearly determine the boundaries of sliding windows and to avoid tuple discards that can occur from windowing tuples (Section III-E).

In this paper, we provide an overview of data stream processing. In what follows, we consider the specifications and processing of stream queries. We also review related work in this area.

## II. QUERY SPECIFICATION

So far, many query languages have been proposed to specify stream queries, including AQuery [11], the box and arrow scheme [1] in Aurora, CQL [5] in STREAM, StreaQuel [7] in TelegraphCQ, and ESL [10] in StreamMill. There are also the special-purpose query languages that are used in Gigascope [8] and Tribeca [12].

In Aurora [1], users construct query plans via a graphical interface by arranging boxes corresponding to query operators, such as selections, joins, and aggregates. The boxes are connected with directed arcs to specify data flows. The boxes can be rearranged in the optimization phase. On the other hand, most other systems including STREAM support SQL-like languages for query specification. The query languages have syntax to specify sliding windows to deal with unbounded data streams. This section focuses on the specification of window queries.

### A. Window Query Model

A real-time data stream is a sequence of data items that arrive in some order (e.g., timestamp order). In the STREAM project [2, 4, 5], assuming a discrete time domain $T$, a stream is defined as follows.

**Definition 1 (Stream).** A stream $S$ is a possibly infinite bag (multiset) of elements $(s, t)$, where $s$ is a tuple belonging to the schema of $S$, and $t \in T$ is the timestamp of the element.
A relation is also defined with the notion of time, which is different from the conventional relation.

Definition 2 (Relation). A relation $R$ is a mapping from $T$ to a finite bag of tuples belonging to the schema of $R$.

To denote a relation at any time instant $t \in T$, $R(t)$ is used. In the window query model, an unbounded stream is translated into finite relations using windows. Suppose that the latest tuple arrives at a certain time $\tau$ for query $Q1$. Then, $R(\tau)$ will consist of tuples whose arrival timestamps are in the interval of $(\tau - 30, \tau]$. The aggregate max will then be applied to $R(\tau)$. In this example, $R(\cdot)$ will be updated, whenever a new tuple arrives.

The relation $R(\cdot)$ can be organized differently, if a different type of window is applied to the query. The window models can be classified according to the following criteria [3, 7].

- **Movement of the window’s endpoints**: Two fixed endpoints define a fixed window, while two sliding endpoints define a sliding window. One fixed endpoint and one moving endpoint define a landmark window.
- **Physical vs. logical**: Physical or time-based windows are defined in terms of a time interval, while logical or count-based windows are defined in terms of the number of tuples. The latter windows are also called tuple-based windows.
- **Update interval**: Eager re-evaluation updates the window whenever a new tuple arrives on a stream, while lazy re-evaluation induces a jumping window. If the update interval is larger than or equal to the window size, the window is called a tumbling window.

If sliding windows are used in a query, the query becomes monotonic. Let $A(Q, t)$ be the answer set of a window query $Q$ at time $t$, and $0$ be the starting time. The query is then re-evaluated over newly arriving tuples, and qualifying tuples are appended to the result. The answer set of $Q$ at time $t$ can be represented as follows.

$$A(Q, t) = \bigcup_{i=1}^{1}(A(Q, i) - A(Q, i-1)) \cup A(Q, 0)$$

On the other hand, if landmark windows are used, the query will be non-monotonic. It will be recomputed from scratch during every query re-evaluation. The answer set of $Q$ at time $t$ can be represented as follows.

$$A(Q, t) = \bigcup_{i=0}^{t-1} A(Q, i)$$

**B. Window Syntax**

To specify a sliding window, the syntax proposed in [6] can be used. In the syntax, a window can be defined with three parameters: 1) RANGE, to denote a window size; 2) SLIDE, to denote a slide interval of the window; and 3) WATTR, to denote a windowing attribute—the attribute over which RANGE and SLIDE are specified.

In general, time-based sliding windows are most commonly used in stream applications [13, 14]. The previous query $Q1$ is an example of a time-based window with a size of 30 seconds. A tuple-based window can also be defined with the RANGE parameter. The following shows a query to compute the max value over the latest 100 tuples.

$$Q2. \text{SELECT MAX(value)}$$
$$\quad \text{FROM Sensors [RANGE 100 rows]}$$

To define an update interval for the window, the SLIDE parameter can be used. The following query updates the max value every 30 seconds, which is calculated over tuples arriving for the latest 30 seconds. The query is an example of the tumbling window.

$$Q3. \text{SELECT MAX(value)}$$
$$\quad \text{FROM Sensors [RANGE 30 seconds, SLIDE 30 seconds]}$$

Note that, with time-based windows, the window interval is determined based on the arrival timestamp of the last input tuple. On the other hand, windowing can also be performed based on different attribute values, not only arrival timestamps. For instance, a user may want to perform the windowing based on the tuples’ generation timestamps. Let the attribute denoting a tuple’s generation timestamp be sourceTS. To use it for windowing, a WATTR parameter can be used as follows.

$$Q4. \text{SELECT MAX(value)}$$
$$\quad \text{FROM Sensors [RANGE 30 seconds, SLIDE 30 seconds WATTR sourceTS]}$$

The following query shows another example of using the WATTR parameter. The query will be posed to identify the list of product names of the latest 10 orders issued from customers.

$$Q5. \text{SELECT productName}$$
$$\quad \text{FROM Orders [RANGE 10 rows, WATTR orderId]}$$

When a windowing attribute is explicitly specified, as in $Q4$ and $Q5$, input tuples may not be in an increasing order of the windowing attribute. This is because input tuples may experience different network transmission delays. Out-of-order input tuples complicate the identification of window boundaries and contents. To simplify this job, existing approaches usually discard out-of-order tuples. These tuple drops may lead to inaccuracy in aggregate queries. This issue will be discussed in Section III-E.
C. Named Queries

To chain a number of relevant queries, the concept of named queries was introduced in STREAM. Consider that we want to identify congested segments on a highway. The highway is divided into a number of segments with exit and entrance ramps at each segment boundary. In this example, each vehicle periodically transmits its speed and position measurements. Suppose the schema of the measurements is \( \text{PosSpeedStr}(\text{vid}, \text{speed}, \text{pos}) \), where \( \text{PosSpeedStr} \) is the name of the input stream. The attributes \( \text{vid} \), \( \text{speed} \), and \( \text{pos} \) denote the id of a vehicle, its speed in miles per hour (MPH), and its position on the highway in feet, respectively.

To calculate the number of vehicles in each segment, the position of a vehicle should first be converted to its corresponding segment number. Assume that segments are exactly 1 mile long. Then, the segment number can be computed by dividing \( \text{pos} \) by 1760. The query for this purpose can be specified as \( Q6 \), named \( \text{SegSpeedStr} \). The name of a query usually comes ahead of the SELECT-FROM-WHERE clause.

\[
Q6. \text{SegSpeedStr} \\
\text{SELECT vid, speed, pos/1760 as segno} \\
\text{FROM PosSpeedStr}
\]

From \( \text{SegSpeedStr} \), the congested segments can be identified. Suppose that a segment is considered congested, if the average speed of vehicles in the segment in the latest 5 minutes is less than 40 MPH. Then, a query to identify congested segments can be constructed as shown in \( Q7 \). For the complete description of this Linear Road Benchmark example, refer to Arasu et al. [5] and Jain et al. [15].

\[
Q7. \text{CongestedSegRel} \\
\text{SELECT segno} \\
\text{FROM SegSpeedStr [RANGE 5 minutes]} \\
\text{GROUP BY segno} \\
\text{HAVING AVG(speed) < 40}
\]

As shown in this example, the outputs of a query \( Q \) can be fed to other queries \( Q_i \) by specifying the name of \( Q \) in the FROM clause of \( Q_i \). Using this scheme, complex application requirements can be specified by chaining a number of simple named queries without introducing a nested query.

III. QUERY PROCESSING

This section discusses the issues in processing of window queries over continuous data streams. The discussion consists of query plan structure, blocking operators, operator scheduling, load shedding, and disorder control.
tuples stored in the ple operators (e.g., tuples through the pointers. If a queue is shared by multi-

Queues are generally organized to have pointers to tuples stored in the memory manager (Fig. 1). An opera-
tor reads the pointers from its input queues and accesses tuples through the pointers. If a queue is shared by multi-

B. Operators

As mentioned earlier, query operators can be stateless or stateful. The behavior of stateless operators is the same

1) Window Aggregates

To compute window aggregates, many existing algo-
rithms use the divide-and-conquer approach. For example,
a dataset X is divided into disjoint subsets X_1, X_2, ..., X_m where X = \bigcup_{1 \leq i \leq m} X_i. Then, an aggregate over X, f(X), is computed using sub-aggregates over X_i as shown in the following equation. Below, g(·) and h(·) are some aggregate functions.

\[ f(x) = h((g(X_i) \mid 1 \leq i \leq m)) \]

For any possible sub-aggregate function g(·), if there is no constant bound on the size of storage needed to store
the result of g(·), the function f(·) is holistic. The examples of holistic aggregates include MEDIAN, QUAN-
TILE, and MODE. Otherwise, the function can again be classified into distributive and algebraic [21]. If g(·) and h(·) are equal to f(·), the function is distributive. Examples include SUM, COUNT, MAX, and MIN. Otherwise, the function is algebraic. AVG is algebraic, since it can be computed using SUM and COUNT.

Li et al. [22] proposed an approach for evaluating

aggregate queries when sliding windows are overlapping. When adjacent windows overlap, a stream is divided into disjoint subsets called panes. The panes are also called basic windows in the literature. The size of a pane can be obtained by \( GCD(V_{R}, V_{S}) \), where \( V_{R} \) is the value of a RANGE parameter, and \( V_{S} \) is the value of a SLIDE parameter in a window specification. Window-level aggre-
gates can then be computed from pane-level sub-aggre-
gates. For example, consider a query to find the maximum bid price for the past 4 minutes and update the result every minute. This query can be described as follows.

Q10. SELECT MAX(bid-price)
FROM Bids [RANGE 4 minutes,
SLIDE 1 minute]

Given Q10, a pane becomes a 1-minute-length subwin-
dow (Fig. 3). The method computes the max bid price for each pane. The max price for each window can then be obtained from the max values of four panes that contrib-
ute to the window. This approach can be applied to the other aggregate functions mentioned above.

Arasu and Widom [23] proposed resource sharing

2) Window Joins

When discussing join algorithms, sliding window equijoins
are most commonly considered [24, 25]. Suppose we
want to join m input data streams over a common attribute A. Let the i-th input stream be \( S_i \), and its window size be \( W_i \) (i.e., the value of a RANGE param-
eter). At time t, a tuple s belongs to a windowed substream

\[ \begin{array}{c}
\text{W1} \\
\text{W2} \\
\text{W3} \\
\text{W4} \\
\text{...} \\
\text{P1} \\
\text{P2} \\
\text{P3} \\
\text{P4} \\
\text{P5} \\
\text{P6} \\
\text{P7} \\
\end{array} \]

Fig. 3. Windows composed of four panes.

http://dx.doi.org/10.5626/JCSE.2013.7.4.220 224 Hyeon Gyu Kim and Myoung Ho Kim
metric hash join, and the symmetric nested loop join. Kang et al. [27] showed that when one stream is
faster than the other, an asymmetric combination of hash join and nested loop join can outperform both the sym-
fications. Subsequently, the join is evaluated whenever a
operator. That idea was developed into AMJoin, proposed by Kwon et al. [28, 29]. AMJoin improves the perfor-
multi-way symmetric hash join operator and showed that the
query plan tree [31]. Hong et al. [32] discussed techniques
for processing a large number of extensible markup lan-
the window join was also used to track the motion of
moving objects, or detect the propagation of hazardous
in a study by Viglas et al. [25]. They proposed MJoin, a
way hash join is faster than any tree of binary join
operators. That idea was developed into AMJoin, pro-
portions of inputs to multiple joins, and the shar-
ing of join computation.

C. Operator Scheduling

Query plans are executed via a global scheduler, which
runs each operator in query plans, to help move tuples
through the plans and produce results. The simplest
pricing scheme is the round-robin, where operators
run in a circular order, and each operator runs until it con-
sames all tuples in its input queue. This method is easy to
implement, and starvation-free, but far from optimal.
In many stream management systems, more intelligent
scheduling techniques have been proposed and used,
including train scheduling [1] in Aurora, eddies [33, 34]
in TelegraphCQ, and chain scheduling [35] in STREAM.
The first two methods focused on improving throughputs,
while the objective of the last was to minimize peak memory size (i.e., peak total queue size).

In Aurora, the contents of inter-operator queues can be written to disk, which is different from STREAM. Therefore, to improve performance of query execution, context-switching between operators should be minimized. For this purpose, their scheduling algorithm (called train scheduling) focused on 1) generating long trains of tuples, 2) processing complete trains at once, and 3) passing them to subsequent operators, without having to go to disk. By batching the processing of tuples through operators, they attempted to reduce I/O overhead.

In TelegraphCQ, query scheduling is conducted by eddies [33]. An eddy is a highly adaptive query processing operator that continuously re-optimizes a query, in response to changing runtime conditions. It does this by treating query processing as the routing of tuples through operators and making per-tuple routing decisions. The cost of making per-tuple routing decisions might be high, which has been asserted by various parties. Regarding this, Deshpande [34] implemented eddies in a PostgreSQL open source database system in the context of a TelegraphCQ project and showed that the overhead of the eddy mechanism was negligible.

The chain scheduling in STREAM focused on minimizing a peak total queue size during query processing. For this purpose, they considered the selectivity of each operator in query plan trees. Consider the following simple example. Suppose we have a query plan with two unary operators, \( O_1 \) and \( O_2 \). \( O_1 \) receives tuples from input queue \( q_1 \) and writes its output to \( q_2 \), which connects to the input of \( O_2 \). Let the selectivity of \( O_1 \) be 20\%, i.e., it consumes \( n \) tuples from \( q_1 \) in one time unit, and introduces \( n/5 \) tuples into \( q_2 \). Also assume that \( O_2 \) takes one time unit to operate on \( n/5 \) tuples. Then, there are two possible scheduling strategies:

- **Round-robin scheduling**: Tuples are processed to completion in the order they arrive at \( q_1 \). Each batch of \( n \) tuples in \( q_1 \) is processed by \( O_1 \) and then \( O_2 \), requiring two time units overall.
- **Selectivity-based scheduling**: If there are \( n \) tuples in \( q_2 \), then \( O_2 \) operates on them using one time unit, producing \( n/5 \) new tuples in \( q_2 \). Otherwise, if there are any tuples in \( q_2 \), then up to \( n/5 \) of these tuples are operated on by \( O_2 \), requiring one time unit.

Suppose we have the following arrival pattern: \( n \) tuples arrive at every time instant from \( t = 1 \) to \( t = 7 \), then no tuples arrive from \( t = 8 \) to \( t = 14 \). On average, \( n/2 \) tuples arrive per unit of time, but with an initial burst. Fig. 5 shows the total size of queues \( q_1 \) and \( q_2 \) under the two scheduling strategies during the burst, where each entry is a multiplier for \( n \). As shown in this example, selectivity-based scheduling is clearly preferable in terms of runtime memory overhead during the burst. Babcock et al. [35] extended this scheme to the *chains* of operators within a query plan (i.e., the groups of operators in query paths), when making scheduling decisions.

### D. Load Shedding

Due to the high arrival rate of tuples, memory may not be enough to run queries. In this case, some portion of input or intermediate tuples can be moved out to a disk or can simply be discarded from memory to shed system load. In data stream management systems, the latter approach is generally adopted, because in many stream applications, fast, approximate results are considered more meaningful than delayed, exact answers.

The load shedder in Fig. 1 continuously monitors memory status. If memory is inadequate, it determines 1) in which query plan operators load shedding occurs and 2) how many tuples should be discarded at that point in the plan. Tatbul et al. [36] discussed the problem of determining these two points. In most cases, tuple drops occur at the entry of query plan trees (e.g., input manager) to maximize the effect of load shedding. In their work, the number of tuples to be discarded is estimated based on quality-of-service (QoS) specifications (e.g., importance of tuple values). Babcock et al. [37] discussed the same problem for aggregate queries. Their method focused on minimizing the degree of inaccuracy introduced in query answers. Al-Kateb and Lee [38] considered load shedding for temporal queries and proposed a new accuracy metric for their load shedding decision.

There has also been substantial research in operator-level load shedding. Many studies focused on load shedding in join operators. When memory is not enough for the join, victims should be chosen from the synopses of a join operator (e.g., hash tables). According to victim selection strategies, load shedding algorithms for window joins can be classified into the following three models.

- **Frequency-based model** [26, 39-42]: The priority of a tuple \( s \) is determined in proportion to the number of tuples in windows, whose value is the same as that of \( s \).
- **Age-based model** [43]: The priority of a tuple is determined based on the time since its arrival, rather than its join attribute values.
- **Pattern-based model** [29]: The priority of a tuple is determined based on its arrival pattern in data.
streams (e.g., the order of streams in which the tuple appears).

Das et al. [26] proposed the concept of semantic load shedding, as opposed to random load shedding. In semantic load shedding, join attribute values are considered to maximize a user-defined similarity measure. They considered the MAX-subset measure, which maximizes the number of tuples in the approximate output of the join. The MAX-subset measure was considered for load shedding in many algorithms [40-42]. Das et al. [26] also proposed two heuristics to determine the priority of tuples in an online join: PROB and LIFE. The former discards the tuple with the lowest probability to join with a partner tuple in the other stream. The latter discards the tuple with the lowest probability of its remaining lifetime and the probability that it joins with another tuple.

On the other hand, Srivastava and Widom [43] showed that the frequency-based model (e.g., PROB) is not appropriate in many applications and proposed an age-based model for them. In their model, the expected join multiplicity of a tuple depends on the time since arrival, rather than its join-attribute value. Their method focused on the memory-limited situation. Many other studies including [26, 40-42] also assumed the memory-limited situation, and then discussed their load shedding algorithms. On the other hand, Gedik et al. [39, 44] emphasized a situation where the CPU becomes a bottleneck (i.e., when an input arrival rate exceeds CPU processing speed), and then proposed load shedding techniques to shed the CPU load.

Kwon et al. [29] considered the load shedding problem for applications where join attribute values are unique, and each join attribute value occurs at most once in each data stream. For these applications, the frequency-based model cannot be used. To resolve this issue, they proposed a new load shedding model, in which the priority of a tuple is determined based on its arrival pattern in data streams (e.g., the order of streams in which the tuple appears).

E. Disorder Control

A data stream is an ordered sequence of data items. The order is important in windowing data streams. In general, window operators assume that tuples arrive in an increasing order of their WATTR values. Out-of-order tuple arrivals are ignored to facilitate the identification of window boundaries [1, 6]. However, tuples may not arrive in the WATTR order, since they often experience different network transmission delays. These out-of-order tuples are discarded in the windowing phase, and such tuple drops can lead to inaccurate results in aggregate queries.

To resolve this issue, input tuples are buffered in the input manager (Fig. 1), until they can be outputted without violating the WATTR order. To determine which tuples can go out, the notion of punctuations [45] can be used. A punctuation $p$ is an assertion indicating that no more tuples with attribute value $p$ will be seen in the future.

Heartbeats [13, 46] can be viewed as special kinds of punctuation, where attributes are timestamps of tuples. Heartbeats can be estimated internally in the system or can be given externally from remote stream sources, such as routers. Ding et al. [47] and Li et al. [6] assumed that heartbeats were externally given, and proposed methods for processing join or aggregate queries gracefully. However, external stream sources may not provide heartbeats in real-world applications. In addition, the heartbeats themselves can be out-of-ordered, when stream sources are in remote locations.

When estimating heartbeats internally, existing approaches generally use the maximum network delay seen in the streams. Let the max delay until time $t$ be $m$. Then, the heartbeat is estimated to $t - m$, and all tuples with generation timestamps smaller than $t - m$ are outputted from the buffer. The $k$-ordering mechanism [48], the skew bound estimation [14], the timestamp mechanism in Gigascope [8, 13], and the ordering mechanism in NiagaraCQ [9] are similar to this approach.

The heartbeat estimation algorithms focus on saving as many discarded tuples as possible. Consequently, they tend to keep the buffer size larger than necessary. For example, the adaptive method proposed by Srivastava and Widom [14] 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 seen for the $W$ interval of time from $t$ (Fig. 6). We can easily see that $m$ is kept large most of the time, since it is determined by the two largest values of network delays seen in the stream in a certain period of time.

To resolve this issue, Kim et al. [46] proposed a method to estimate the buffer size based on stream distribution parameters monitored during query execution, such as tuples’ interarrival times and their network

Fig. 6. Estimation of the max delay proposed by Srivastava and Widom [14].
delays. In particular, their method supports an optional window parameter DRATIO (an abbreviation for a drop ratio), to enable a user to control disorder according to application requirements.

Q11. SELECT MAX(value) FROM Sensors [RANGE 30 seconds, DRATIO 1%]

The query above specifies that the percentage of tuple discards permissible during query execution should be less than or equal to 1%. By specifying DRATIO, a user can control the quality of query results according to application requirements; a small value for the drop ratio provides more accurate query results at the expense of high latency (due to a large buffer), whereas a large value gives faster results with less accuracy (from a smaller buffer).

IV. FINAL REMARKS

In this survey, we have tried to coherently present the major technical concepts for data stream processing. To keep the task manageable, we restricted the scope of this paper to the specification and processing of window queries. This is meaningful, because most major stream processing engines, such as Borealis [49] (the successor of Aurora), STREAM, TelegraphCQ, and StreamMill, support an SQL interface, to receive application requirements in the form of window queries [50]. The discussions in this paper included synopses structures, plan sharing, blocking operators, operator scheduling, load shedding, and disorder control.

Many interesting issues related to data stream processing could not be included in this survey, for example:

- Classification and clustering
- Frequent pattern mining
- Synopsis structures, such as reservoir samples, sketches, wavelets, and histograms
- Indexing streams
- Multi-dimensional analysis of data streams

Some of the issues were discussed in other papers. For example, Gaber et al. [51] presented a survey related to mining data streams. Aggarwal and Yu [52] provided a survey on synopsis construction in data streams. Mahdiraji [53] provided a survey on the algorithms for clustering data streams. These surveys can augment the discussions given in this paper.

ACKNOWLEDGMENTS

This research was supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program(NIPA-2013-H0301-13-4009) supervised by the NIPA(National IT Industry Promotion Agency).

REFERENCES


Hyeon Gyu Kim
Hyeon Gyu Kim is currently an assistant professor at the Division of Computer, Sahmyook University, Korea. He was a chief research engineer at LG Electronics for 1 year from 2010 to 2011, and a senior researcher at the Korea Atomic Energy Research Institute for 1 year from 2011 to 2012. He received his B.S. and M.S. degrees in Computer Science from Ulsan University, and his Ph.D. degree in Computer Science from KAIST in 2010. His research interests include data stream processing, databases, and probabilistic safety assessment.

Myoung Ho Kim
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 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.