Storage-centric Load Management for Data Streams with Update Semantics

Technical Report #620

ETH Zürich, D-INFK, 03 2009

Alexandru Moga, Irina Botan, Nesime Tatbul

Systems Group, Department of Computer Science, ETH Zürich

{*amoga,irina.botan,tatbul*}*@inf.ethz.ch*

March 2009

Abstract

Most data stream processing systems model their inputs as append-only sequences dfg of data elements. In this model, the application expects to receive a query answer on the complete input stream. However, there are many situations in which each data element (or a window of data elements) in the stream is in fact an update to a previous one, and therefore, the most recent arrival is all that really matters to the application. UpStream defines a storage-centric approach to efficiently processing continuous queries under such an update-based stream data model. The goal is to provide the most up-to-date answers to the application with the lowest staleness possible. To achieve this, we developed a lossy tuple storage model (called an "update queue"), which under high load, will choose to sacrifice old tuples in favor of newer ones using a number of different update key scheduling heuristics. Our techniques can correctly process queries with different types of streaming operators (including sliding windows), while efficiently handling large numbers of update keys with different update frequencies. We present a detailed analysis and experimental evidence showing the effectiveness of our algorithms using both synthetic as well as real data sets.

Contents

1 Introduction

Processing high-volume data streams in real time has been a challenge for many applications including financial services, multi-player online games, security monitoring and location tracking systems. Various load management techniques have been proposed to deal with this challenge from dynamic load balancing (e.g., [29]) to adaptive load shedding (e.g., [25]). Most of these techniques are best-effort in nature and rely heavily on application-specific resource allocation and system optimization techniques based on Quality of Service (QoS) specifications. In UpStream, we also deal with the load management problem for real-time streaming applications and take an application-specific approach, but focusing on a different property essential to a common set of applications: *update semantics*.

Most current stream processing systems model their inputs as append-only sequences of data elements. In this model, the output streams that are delivered to the end-point application are also interpreted as append-only, and therefore, the application expects to receive a query answer on the complete input stream. However, there are many cases in which each data element (or a window of data elements) in a stream in fact represents an update to a previous one, and therefore, *the most recent arrival* is all that really matters to the application. For example, a stock broker watching a continuously updating market dashboard might be interested in the *current* market value of a particular stock symbol. As another example, in a facility monitoring system, one might like to watch for the *latest* 5-minute average temperature in all rooms. Similarly, in systems that involve continuous mobile object tracking such as in Intelligent Transportation Systems, there is a need to monitor the *current* GPS location of each vehicle as well as the *latest* average vehicle speed or traffic flow for selected road segments [1]. In such applications with update semantics, the main goal is to provide *the most up-to-date answers* to the application with *the lowest staleness* possible, as opposed to streams with append semantics, where providing *all answers* with *the lowest latency* is more important.

Latency and *staleness* are related, but different quality metrics. Latency for a data element captures the total time that it spends in the system (queue waiting time + processing time) before it is made available to the output application, and therefore, is a quality metric that is attributed to individual output tuples. On the other hand, staleness is a metric that is attributed to the output as a whole and is determined by subsequent input arrivals that affect that output. An output will have zero staleness as long as it reflects the result that corresponds to the latest input arrivals. However, as soon as a newer arrival occurs that would invalidate that result, the staleness of that output starts going up.

Under normal load conditions where the system can keep the processing up with data arrival rates, staleness is a direct result of latency. In this case, staleness of an output can be minimized by minimizing the latency of its output tuples. For this reason, append and update semantics in effect produce similar results for the application. However, under overload, the situation is different. In this case, if no load is shed, latency for a stream monotonically grows, and so does its staleness. For streams with append semantics, various load shedding techniques have been proposed to deal with the latency issue (e.g., [25, 7, 22, 26, 27, 24]). These techniques necessarily result in inaccurate query answers, and the focus has therefore been on minimizing the degree of this accuracy loss. However, none of these techniques take update semantics into account, and therefore can not ensure a decrease in staleness. Furthermore, accuracy loss is not a significant issue for applications with update semantics since they are not interested in receiving all the values anyway. In this respect, update streams naturally lend themselves to load shedding. What is currently missing is a resource-efficient framework that ensures that this is done in a systematic way to directly minimize staleness while providing the desired update semantics to the application.

UpStream proposes *a storage-centric framework* for efficiently processing continuous queries under an update-based stream data model. As discussed above, the primary goal is to provide the most up-to-date answers to the application with the lowest staleness possible at all times. To achieve this, we developed a lossy tuple storage model (called an “update queue”), which under high load, will adaptively choose to sacrifice old tuples in favor of newer ones. Our technique can correctly process queries with different types of streaming operators including ones with sliding windows, while efficiently handling large numbers of update keys in terms of minimizing staleness and memory consumption.

In this report, we would like to introduce the following contributions of UpStream:

- a new update-based data stream processing model,
- a storage-centric load management framework for update streams based on update queues
- techniques for correctly and efficiently processing update streams across complex sliding window queries in order to minimize staleness and memory consumption.

The rest of this report is organized as follows: We first summarize the related work in Section 2. In Section 3, we describe the basic models and definitions that set the stage for the research problem that UpStream addresses. Section 4 introduces our storage-centric load management framework. Section 5 describes some key scheduling methods for minimizing staleness. Section 6 explains how our framework can process queries with sliding windows in a way to ensure correctness and low staleness, while using the memory resources efficiently. In Section 7, we present the implementation of our techniques, putting all the pieces together. Section 8 contains a thorough experimental performance analysis on a prototype implementation that builds on the Borealis stream processing system [3].. Finally, we conclude with a discussion of future work in Section 9.

2 Related Work

Our work on update streams mainly relates to previous work in the following research areas:

Stream Load Management. The existing work in stream load management treats streams as append-only sequences and therefore focuses mainly on minimizing latency. Two classes of approaches exist. The first class focuses on load distribution and balancing, while the second class focuses on load shedding. Load distribution and balancing involves both coming up with a good initial operator placement (e.g., [28]) as well as dynamically changing this placement as data arrival rates change (e.g., [8], [29], [19]). In general, moving load is a heavy-weight operation whose cost can only be amortized for sufficiently long duration bursts in load. For short-term bursts leading to temporary overload, load shedding is proposed. In load shedding, the distribution of operators onto the processing nodes is kept fixed, but other load reduction methods (e.g., drop operators, data summaries) are applied on the query plans which results in approximate answers (e.g., [25], [7], [22], [26], [27], [24]). All of these techniques focused on reducing latency for applications with append semantics, and none of them provided storage-based solutions.

Synchronization and Freshness in Web Databases. Cho and Garcia-Molina study the problem of update synchronization of local copies of remote database sources in a web environment [12]. The synchronization is achieved by the local database polling the remote one, and the main issue is to determine how often and in which order to issue poll requests to each data item in order to maximize the time-averaged freshness of the local copy. In our problem, updates from streaming data sources are pushed through continuous queries to proactively synchronize the query results. Since the exact update arrival times are known, this gives our algorithms direct control for synchronization. In a more recent work by Qu et al, two types of transactions for web databases have been considered: query transactions and update transactions [21, 20]. To provide timeliness for the former and freshness for the latter, an adaptive load control scheme has been proposed. The update transactions in this line of work have a very similar semantics to our update streams. However, a major difference is our push-based processing model: we do not separate query and update transactions, but rather, consider updates and queries as part of a single process due to the continuous query semantics. Finally, Sharaf et al propose a scheduling policy for multiple continuous queries in order to maximize the freshness of the output streams disseminated by a web server [23]. This work only focuses on filter queries. Furthermore, it is assumed that occasional bursts in data rates are short-term and all input updates are eventually delivered (i.e., append semantics). In our work, we focus on update semantics, where delivering the most recent result in overload scenarios is the main requirement.

Materialized View Maintenance. Previous work on materialized view maintenance has also close relevance to our work. The STanford Real-time Information Processor (STRIP) separates view imports from view exports. In this model, both the base data as well as the derived data materialized as views must be refreshed as updates are received (view import). Also, read transactions on both the base data as well as materialized views must be executed, with specific deadlines (view export). As in web databases, this creates a tradeoff between response time for read transactions and freshness for update transactions. Adelberg et al propose scheduling algorithms for efficient updates on base data [4], as well as on derived data [5]. Kao et al further extend these works by proposing scheduling policies for ensuring temporal consistency [17]. A more recent and closely related work to UpStream is the DataDepot Project from AT&T Labs [14], [13], [9]. DataDepot is a tool for generating data warehouses from streaming data feeds, thus it has many warehousing-related features. For us, the part on real-time update scheduling is directly relevant. We see two basic similarities between UpStream and DataDepot: both accept push-based data and both worry about staleness. On the other hand, in DataDepot, updates correspond to appending new data to warehouse tables. Therefore, all updates must be applied. Furthermore, DataDepot focuses on scheduling the update jobs, but does not consider continuous operations on streams (e.g., sliding window queries). UpStream could potentially serve as a pre-processor for a real-time data warehouse system such as DataDepot. Thus, the

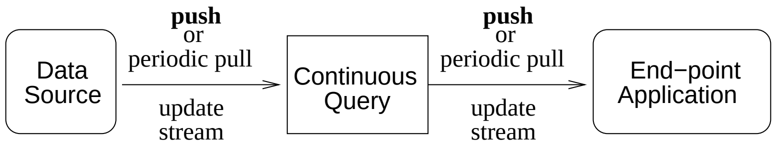


Figure 1: Continuous Query Models

two works are complementary.

3 Models

In this section, we describe the basic models and assumptions that underlie our work and define the problem addressed in UpStream.

3.1 Data Model

We model data streams as totally ordered sequences of tuples (usually by time), where each tuple represents an update to the one before it in the sequence. Furthermore, one or more fields in the stream schema can be designated as the *update key*. If this is the case, a tuple updates a previous tuple with the same key value and the ordering requirement has to only hold among the tuples that carry the same key value. For example, given a stream of stock prices in the form of (time, $\underline{\text{symbol}}$, price) tuples, the symbol field represents the update key. In this case, (09:01, IBM, 25) is an update on an earlier tuple (9:00, IBM, 24).

To generalize, an *update stream* consists of a sequence of relational tuples with the generic schema (time, $\underline{\text{update-key-fields}}$, other-fields). We assume that tuples are ordered by time for a given update key and this per-key order is preserved throughout the query processing. Furthermore, we assume that update key fields are retained in the tuples throughout the query plan. This enables the user and the system to understand the connection between the input tuples and the query result tuples that belong to the same update key value.

3.2 Query Processing Model

Continuous queries are defined on update streams. A continuous query takes an update stream as input, and produces an update stream as output. Depending on the mode of input and output data delivery, there are four possible continuous query models. Figure 1 illustrates these alternatives. UpStream focuses on the push-push model.

As new update tuples arrive from the sources, the continuous queries need to be re-evaluated, refreshing a previously reported query result to reflect the most recent state of the source data. Until new query results become available, the application will rely on the most recently reported result. Therefore, updates have to be processed as soon as possible. Otherwise, the application may perform actions based on a value which is in fact stale and therefore an inaccurate reflection of the real world event that it represents.

Unlike append streams, where the application expects to receive query results on all incoming tuples, in the case of update streams, the application needs to be updated only on the most recent values available from the data sources. Therefore, at any point in time, a query result which is produced by using the most up-to-date input values is sufficiently correct compared to a query result which is produced by using all of the available input values ever existed so far. This is a radically different semantics than that in the case of append streams.

As in the case of append streams, continuous queries on update streams can be composed of a number of stream-oriented operators [15]. UpStream focuses on two representative classes of streaming operators: tuple-based operators and sliding window-based operators. The former class includes the stateless operators that execute on a per-tuple basis (e.g., filter, map), whereas the latter class includes stateful operators that execute on a per-window basis (e.g., sliding window aggregate). Tuple-based operators have the same basic form and semantics as their counterparts in the append-based data model with one small exception: They are allowed to apply any arbitrary transformation functions on their input tuples, as long as they retain the update key value in the corresponding output tuples. Window-based operators (such as the sliding window aggregate operator) logically group a given input stream into substreams by their group-by fields and construct sliding windows on each substream based on a given window size (w) and a window slide (s). Then an aggregate function is applied on each such window, producing an output tuple. In case of update streams, the group-by fields must be a superset of the update key fields to be able to retain them in the output [1]. We would like to note here that, the window-based operators essentially map updates on input tuples into updates on windows, each corresponding to an output tuple. This difference in operational unit must be taken into account in managing the updates in the system, since output staleness will apply to whole windows rather than individual input tuples. This is an important issue that we will discuss in detail in Section 6.

To present, our work includes a thorough study of update semantics for the above mentioned classes of operators. As a further step towards completeness of the query processing model, we will investigate other types of streaming operators, like n-ary operators:

- *Union:* Union is an n-ary operator that merges n input streams into a single output stream. In case of update streams, all inputs should not only have the same schema, but also the same update key fields. Additionally, the merge must be order-sensitive in that the total time order of the inputs for each update key must also be preserved at the output. Furthermore, since we assume that update streams have total order on time for a given update key, Union is not allowed to produce multiple output tuples with the same time value for the same update key. For example, assume a Union with two inputs A and B, and that input A contains (9:00, IBM, 24). If input B also contains a tuple for IBM at time 9:00, then Union must only keep one of these tuples, and the other one should be dropped [2]. It is

[1]In this report, for ease of presentation, we simply assume that the group-by fields are the same as the update key fields. Our techniques are general enough to handle the more general superset case.

[2]Note that if the ”other-fields” (in this example, price) carry the same content, then it does not matter semantically, which tuple is kept and which one is dropped. If not, we would consider the input to

important to note that the order requirement would cause Union to block in the case of append streams. However, in the update streams case, it would be sufficient to deliver the most recent update. Therefore, Union can deliver the youngest tuple in any of its input queues at the time of processing, and then can simply drop the subsequently arriving tuples on the other input queues that might have smaller time values than the last delivered one.

- *Join:* Join is a binary operator that correlates two input streams based on a window size and a predicate. Tuple pairs from two streams that coincide in the same window are checked for the predicate, and the ones that satisfy the predicate are concatenated to produce an output. In the case of update streams, in order to retain the update key fields in the output, the join predicate is assumed to implicitly include a condition on the equality of the key fields received from both join inputs. Additional join conditions can be added as a conjunction to this equality predicate. Note that it would be sufficient to deliver the most recent match if there are multiple matching pairs in the join window.

3.3 Quality of Service Model

In append-based streams, end-to-end processing latency is the most important QoS metric. Since all input tuples are expected to be processed completely, the main focus is on minimizing the time that each tuple spends in the system until it is processed and a corresponding result is appended to the output stream. In the case of update streams, it is more important to deliver the most current result than to deliver all results with low-latency. Thus, the focus is on making sure that a given query result is as much up-to-date as possible with respect to the consequent updates arriving on its input streams. Thus, we use a different QoS metric, *staleness*, to capture this goal.

Staleness metric was also used by previous work for bringing multiple data copies up-to-date in various different contexts including cache synchronization and materialized view maintenance [6, 12, 18, 17]. The exact definition varies depending on the problem context and is usually based on one of the following: time difference, value difference, or number of unapplied updates. In our current work, we use a time-based staleness definition. Staleness is a property that we associate with each output substream that we deliver for each different update key value through each continuous query. It shows how much a result tuple with a certain update key k falls behind in time with respect to more recent input arrivals for k. Figure 2 illustrates our definition. In this example, there is one continuous query and a single update key value. Input tuples (shown with letters) arrive at a rate of 1 update per 2 time units, while the query has a processing cost of 3 time units per update. As soon as a new input update arrives, the staleness of the output starts increasing with time. As soon as a new result is delivered, staleness drops. If there are no newer input updates, then the staleness goes to zero. Otherwise,

be illegal since an update key can not be associated with multiple values at a given time point (e.g., IBM's price can not be both 24 and 25 at 9:00). If such conflicts are expected to occur for the data sources involved in an application, then the application can model that into Union implementation by stating which input source should be taken as the truth. In general, for the sake of freshness, it would be preferable to keep the input tuple that is readily available to Union at the time of processing.

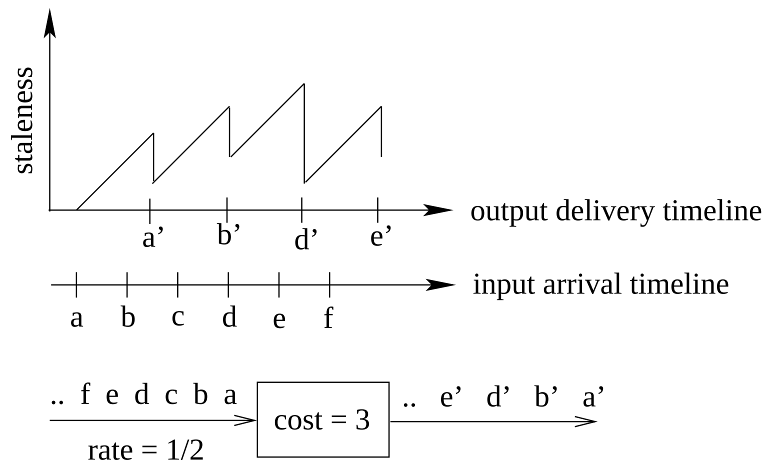


Figure 2: Staleness

it goes down by an amount based on the first invalidation time point where a newer update was received. For example, tuple b arrived while tuple a was being processed. When a' is delivered, staleness only goes down to (now - arrival time for b).

We can define staleness more precisely as follows. Assume an input stream I processed through a query Q, with N distinct update key values. For each output substream $O_i, 1 \leq i \leq N$ with update key value k_i, let $s_i(t)$ denote the staleness value at time t. Then:

$$s_i(t) = \begin{cases} 0, & \text{if there are no pending updates on } I \text{ with } k_i \\ t - \tau, & \text{where } \tau \text{ is the arrival time of the oldest} \\ & \text{pending update on } I \text{ with } k_i \end{cases}$$

The average staleness of O_i over a time period from t_1 to t_2 is:

$$s_i = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} s_i(t)dt$$

Our overall goal is to minimize the average s_i over all output substreams of all queries in the system during a given run time period.

It is important to emphasize here that, for window-based queries, the output staleness applies to the results of whole windows rather than individual tuples. Thus, in the above staleness definition, "pending update" would then correspond to the result of a fully arrived window, i.e., windows that have only partially arrived in a system would not cause an increase in output staleness. An example is given in Figure 3 where a simple Aggregate forms windows of size 5 sliding by 2 over the input stream. On the upper part of the figure we can see how staleness evolves. Output results are indicated by the delimiters of the window (e.g.: [5,9] means the window that opened at 5 and closed at 9).

The processing cost for a window (that is, for a full update) is expressed differently than in

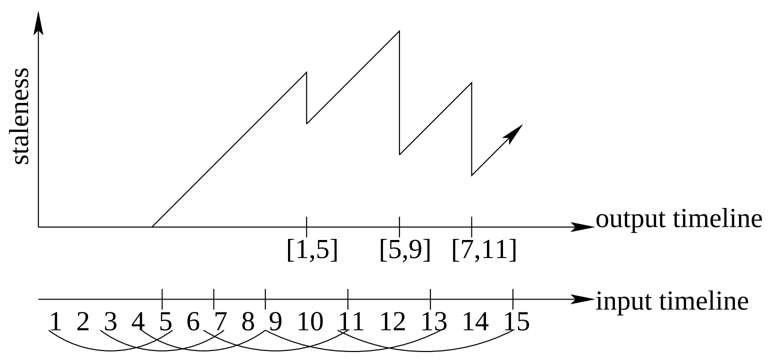


Figure 3: Staleness for sliding windowing operations.

the tuple-based scenario. Here, the cost for updating the application C must account for (i) the cost incurred by the aggregation query, C_{aggr}, to dequeue and process the window tuples and (ii) the cost for computing the aggregation results and delivering them, C_w. C_{aggr} is given by a number of factors like the processing cost per tuple C_t, the number of tuples within the window (w) or the degree of overlapping between windows ($o = [\frac{w}{s}]$). Ideally, C_{aggr} is modeled by the following formula:

$$C_{aggr} = C_t * \sum_{i=0}^{o-1} (\omega - i * \delta)$$

In the case of window-aware load shedding, i can vary between 0 and $(o - 1)$. This yields:

$$C_t * \omega \leq C_{aggr} \leq (o + 1) * C_t * (\omega - \frac{o}{2} * \delta)$$

Because of this variation, for windowing update semantics we decided to provide an average cost estimation for C_{aggr}. Assuming $\overline{C_t}$ is an average cost per tuple spent by the aggregation query, then:

$$C = \overline{C_t} * w + C_w$$

3.4 Load model

In the previous section we have shown our staleness-based QoS model. Now it is time to identify the parameters that could influence staleness.

First of all, when the system can process incoming tuples with acceptable QoS (staleness) levels, we say that the system is underloaded (within its capacity). The most basic form of overload is directly related to input rates. The ratio between the processing cost (C) and the input rate ($\frac{1}{TBA}$, where TBA stands for time between arrivals) gives us a theoretical load to the system, which from now on we will call *the load factor* ($LF = \frac{C}{TBA}$).

Under update semantics, the system load can be broken down in two dimensions:

1 The first is query load and is given by the *number of keys* that have pending updates. This load dimension is similar to how a CPU load is modeled. That is, a key having pending updates is equivalent to a process waiting for the CPU. Besides this, in the window-based processing case, we have another aspect for query load. Since an update is marked by the end of a window and the window slide determines where windows close in the stream, varying the window slide would also introduce load for the query.

2 The second dimension refers to the key update rates indicating how fast tuples belonging to a certain key arrive in the system. If we consider 10 keys updating uniformly within a certain interval at an input rate of 1000 per second then: (i) the *combined update rate*, or simply the input rate, is 1000 per second (ii) the average *key update rate* is 100 per second. Increasing the input rate also increases the key update rate. Having more keys updating in the same interval at the same combined rate will decrease the key update rate.

3.5 Problem Definition

Based on the models we introduced above, we can define our general research problem as follows. We are given a set of continuous queries with update semantics, represented as a plan of operators. This query plan takes a set of update streams as input and produces a set of update streams as output. Each update stream may also have an update key, where each key may be updating at different rates over the course of system execution. Given this setup, implement an adaptive processing framework so as to minimize the average staleness of the result for each update key over all continuous queries in the system. At the moment, we focus on solving this problem for a single continuous query. Extending our solution to the scheduling of multiple queries involves a number of orthogonal issues, which we are planning to address as part of our future work.

4 Storage-Centric Approach

In this work, we take a storage-centric approach to load management for streaming applications with update semantics. Our motivation for doing so is threefold. First of all, storage is the first place that input tuples hit in the system before they get processed by the query processing engine. More specifically, input tuples are first pushed into a tuple queue, where they are temporarily stored until they are consumed. The earlier the update semantics can be pushed in the processing pipeline, the better it is for taking the right measures for lowering staleness. Second, it is rather easy and efficient to capture update semantics as part of a tuple queue. For example, applying “in place updates” in a queue would be rather straight-forward and also memory-efficient. Finally, a storage-based framework allows us to accommodate continuous queries with both append and update semantics in the same system, by defining their storage mechanisms accordingly. In other words, one can selectively specify certain tuple queues as “append queues”, whereas others as “update queues” without making any changes in the query processing engine. This kind of a model is also in agreement with recently proposed streaming architectures that decouple storage from query processing such as the SMS framework [10].

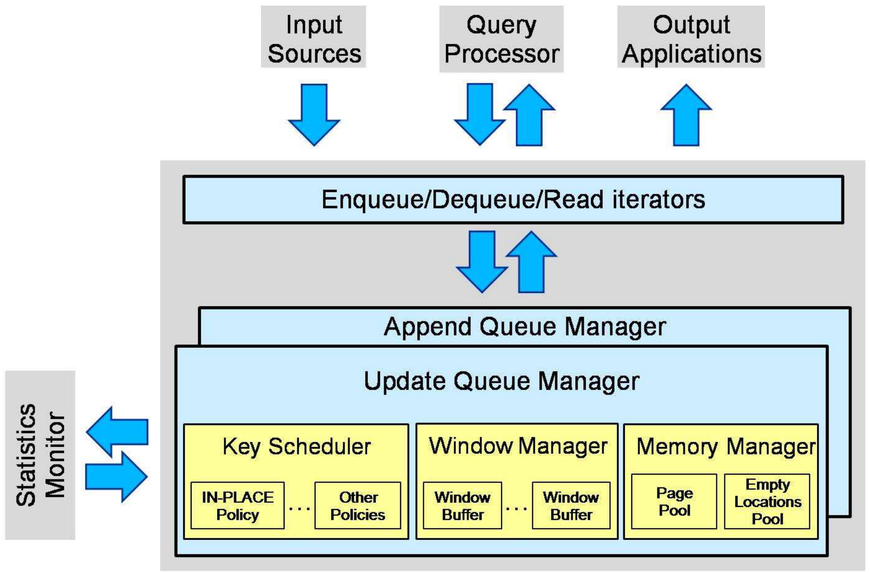


Figure 4: UpStream Storage Manager Architecture

In what follows, we first introduce the concept of update queues; then we describe the design of our storage manager architecture.

4.1 Update Queues

Traditional append-based query processing model requires tuple queues be modeled as append-only data structures with FIFO semantics. To support the new update-based query processing model, we have extended the traditional model with *update queues*. The main property of an update queue is that, for each distinct update key value, it is only responsible for keeping the most recent tuple model; older ones can be discarded [3]. One can think of update queues as having one data cell per update key value, which is overwritten by newer values at every arrival. Given this basic property, an update queue can still be implemented in different flavors based on how update keys are ordered, how the underlying in-memory data structures are managed, and how windowing semantics for sliding window queries is taken into account at the queue level. We separate these issues into three orthogonal dimensions in our storage framework. Next we describe the storage manager architecture that encapsulates this design.

4.2 Storage Manager Architecture

Figure 4 shows the architecture of our storage manager. Storage manager interfaces with input sources, output applications, and query processor through its iterators. These iterators enable

[3]For ease of presentation, we are simply discussing tuple-based queries here; extensions to window-based queries will be provided later in Section 6.

three basic queue operations: enqueue, dequeue, and read. Input sources always enqueue new tuples into a queue, whereas output applications always dequeue tuples from a queue. Query processor can enqueue intermediate results of operators, while it can read or dequeue these back again to feed into the next operators in the query pipeline. Storage manager also communicates with the statistics monitor in order to get statistics to drive its optimization decisions. The underlying queue semantic can either be an append queue or an update queue. Here, we focus on the latter.

The update queue manager itself is broken into three main components. The Key Scheduler (KS) decides when to schedule different update keys for execution and can employ various different policies for this purpose (in-place updates, etc.). The Window Manager (WM) takes care of maintaining the window buffers according to the desired sliding window semantics. Finally, the Memory Manager (MM) component oversees the physical page allocation from the memory pool. In our design, these three components are layered on top of each other and handle three orthogonal issues: KS is responsible for minimizing staleness, WM is responsible for correct window processing, and MM is responsible for the efficient management of the available system memory where the actual data is physically stored. Next, we will describe how we handle these issues.

5 Minimizing Staleness

In this section, we focus on the update key scheduling component of the UpStream storage framework. This component directly deals with minimizing the overall average output staleness.

5.1 Application Staleness vs. Queue Staleness

As presented in Section 3.3, computing application-perceived staleness involves keeping track of several system events including successive arrivals into an update queue, actual deliveries to the end-point application, and the relative order of the occurrence of these events. All run-time components of a stream processing system can influence these events, and therefore, the actual application-perceived staleness. In UpStream, our goal is to be able to control staleness at the storage level. In order to facilitate this, we introduce a new staleness metric called *queue staleness*, which is simpler than the application-perceived staleness (let us call it *application staleness* hereafter), yet can capture the essence of staleness while it can also be directly measured and controlled at the storage level. This can greatly simplify our storage optimization.

Queue staleness for an update key is determined based on how long that key waits in the queue from the first enqueued update until the next dequeue for that key. Thus, the actual processing cost is factored out of the application staleness. Next we show that one can control application staleness by controlling queue staleness.

Figure 5 shows two staleness plots for a single update key: (i) solid line for application staleness, (ii) dashed line for queue staleness. The x-axis represents the run time. As seen, updates first get enqueued into the update queue, then dequeued for being processed, and finally

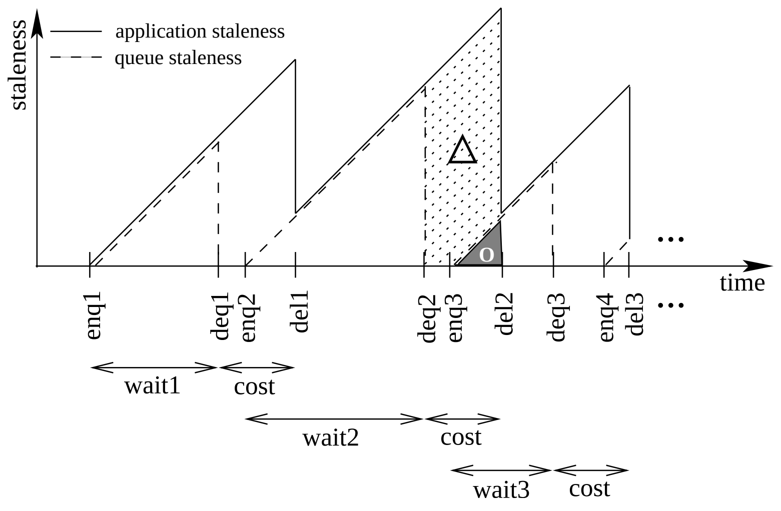


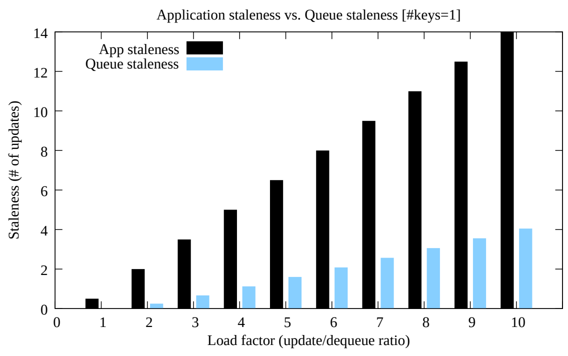
Figure 5: Application Staleness vs. Queue Staleness

get delivered to the application. The average staleness would be computed by taking the area under a given plot and then dividing it by the total run time. Instead of considering a whole run, we will focus on a representative time interval that will repeat itself at steady state, namely the middle part of Figure 5. For this part of the plot, we can compute the area difference between the application and queue staleness plots (i.e., the dotted area denoted by Δ) as follows:

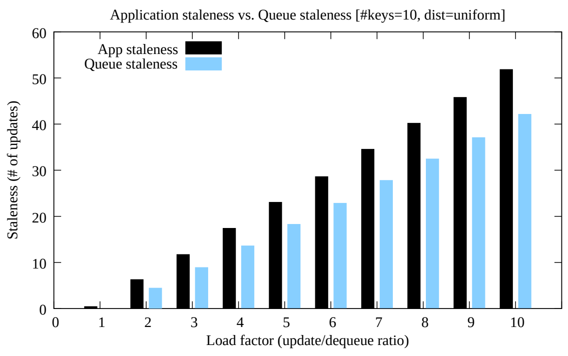
$$\begin{aligned}\Delta &= \frac{(wait + cost)^2}{2} - \frac{wait^2}{2} - O \\ &= wait * cost + \frac{cost^2}{2} - O\end{aligned}$$

where $wait$ is the average queue waiting time (i.e., average over wait1, wait2, wait3, etc. shown in the figure); $cost$ is the average query processing cost; and O is the overlap area that actually belongs to the next iteration, and thus must be subtracted. Due to the overwriting behavior of the update queue, O is bounded as follows: $0 \leq O \leq \frac{cost^2}{2}$. Therefore, $wait * cost \leq \Delta \leq wait * cost + \frac{cost^2}{2}$. In other words, the difference between two staleness metrics depends only on $wait$ and $cost$. Since $cost$ is a constant and $wait$ directly determines the queue staleness, we can conclude that minimizing queue staleness also minimizes application staleness. Thus, we can focus on the queue staleness metric for our analyses and optimizations. Please note that the fact that we allow queue waiting time to vary and use an average queue waiting time in the equation makes is possible to generalize this result to the case for multiple update keys.

We have also experimentally verified our above analysis using a simulator of the Key Scheduler (see Figure 7). In such a simulator, we model an update queue purely as a queue of keys, not having to worry about the underlying storage, query scheduling semantics, etc. The order of the queue residents is decided by various key scheduling strategies which we will discuss in the



(a) Single key



(b) Multiple keys

Figure 6: Application staleness vs. Queue Staleness

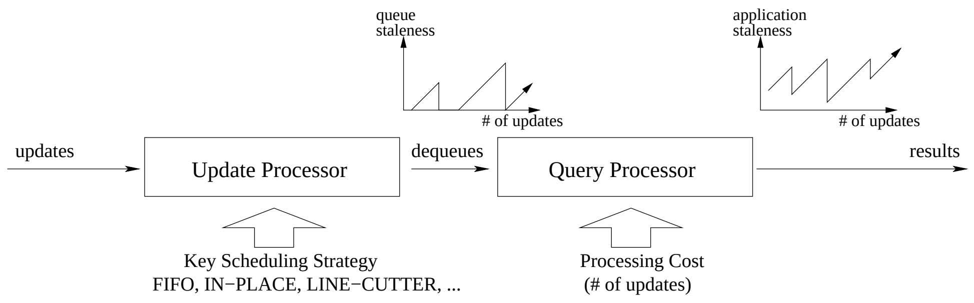
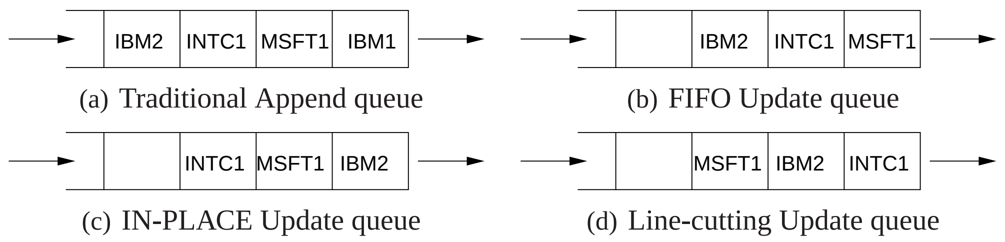


Figure 7: Key scheduling Simulator

following two sections. In our simulator, staleness is computed in terms of *number of updates*. Every tuple on the input stream is considered an update for a certain key. The processing cost associated to a key is also expressed in number of updates. The Update Processor contains the queue of keys and applies the specified key scheduling strategy. The Query Processor simulates a query that applies a processing cost to tuples. Applications staleness is computed for output tuples and queue staleness is computed for dequeued tuples.

Figures 6(a) and 6(b) compares the two staleness metrics for a single key and a multiple key setup, respectively. We show how overall average staleness measured in units of number of updates scales with increasing load. In both cases, staleness grows in a similar way and the difference between the two staleness metrics is determined by the queue waiting time, which is solely a linear function of the load factor. Please note that compared to Figure 6(a), in Figure 6(b), the queue staleness is inflated by about a factor of 10, since there are 10 distinct update keys that have to wait for each other in the queue, and this increases the queue waiting time by a factor of 10. However, the number of keys does not affect the difference between application and queue staleness, which agrees with our theoretical analysis.

5.2 FIFO and IN-PLACE Update Queues



(a) Traditional Append queue (b) FIFO Update queue

(c) IN-PLACE Update queue (d) Line-cutting Update queue

Figure 8: Update queue examples

We now introduce two baseline approaches to update key scheduling: FIFO and IN-PLACE.

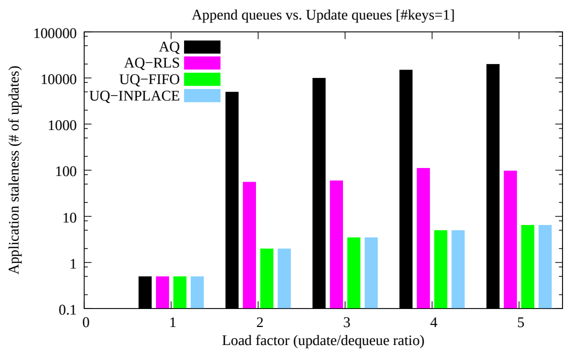
- **FIFO Key Scheduling:** A FIFO update queue is one where tuples are serviced in their arrival order, both for a given key and across all keys. Its main difference from a FIFO append queue is that it is lossy: once a new value for an update key arrives, it is inserted at the tail of the queue while the older value for the same key, if already in the queue, is removed. In the example shown in Figure 8(b), arrival of a new tuple for IBM (IBM2) at the tail of the queue removes the old IBM tuple (IBM1) from its earlier position in the queue.
- **IN-PLACE Key Scheduling:** An IN-PLACE update queue is one where tuples are serviced in FIFO arrival order within a key group, but not necessarily so across different key groups. Once a tuple with a certain key value gets into the queue, it preserves its position in the queue even if a later value for that key arrives and overwrites the older one. In other words, the old value acts like a *place-holder* for the new value. In Figure 12, IBM2 overwrites IBM1 on its original place in the update queue. Unlike the FIFO update queue, a key group in an IN-PLACE update queue does not waste the time it already spent waiting in the queue if it gets superceded by a newer value. This way, key groups can reduce their queue staleness. Thus, compared to a FIFO update queue, an IN-PLACE update queue is already expected to provide a staleness-improving optimization beyond the basic update queue functionality.

We compared the performance of FIFO and IN-PLACE update queues (UQ-FIFO and UQ-INPLACE, respectively) against traditional FIFO append queue (AQ) and its random load shedding counterpart (AQ-RLS) using our simulator [4]. Figure 10 shows that these baseline key scheduling policies significantly improve average staleness.

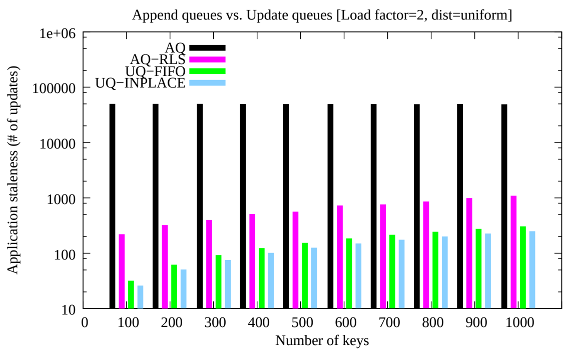
Next we show that, in fact, IN-PLACE policy results in the lowest average staleness that can be achieved when the update frequencies of the update keys are uniformly distributed.

Consider n update keys, each updating uniformly at the same expected frequency value. In other words, we assume that arrival of each update key k_i simply follows a Bernoulli probability distribution with parameter p_i, and that for all keys $k_i, 1 \leq i \leq n$, we are given that $p_1 = p_2 =$

[4]Some of these experiments will be reconsidered in Section 8 where we will observe the behavior of the fully-fledged update queue in a real stream processing deployment.



(a) Single key



(b) Multiple keys

Figure 9: Append queue vs. Update queue

$\ldots = p_n$. Thus, for a run of m updates, we expect $p_i * m = \frac{1}{n} * m$ of them to be for key k_i. Let e_i denote the very first enqueue time for k_i (i.e., more than one updates for k_i may have arrived and overwritten that first update, but we only care about the first dequeue time since that determines the queue staleness). Without loss of generality, assume that $e_1 < e_2 < \ldots < e_n$. Then at any time point t, the queue staleness for these keys must be such that $s_1 > s_2 > \ldots > s_n$. When we dequeue key k_i for processing, then the queue staleness of that key s_i becomes 0 until the next enqueue on k_i occurs. In the mean time, while k_i is being processed, for all $j, 1 \leq j \leq n, j \neq i$, s_j will grow by an amount of $cost$, causing the queue staleness area for that key to grow by an amount of $\frac{s_j+(s_j+cost)}{2} * cost$. To minimize the total area growth for all the undequeued keys, k_i with the highest s_i must be chosen for dequeue. In other words, we must choose the key with the earliest first enqueue time e_i (i.e., in our scenario, k_1 must be chosen). This argument applies independent from how soon the next enqueue for the chosen key occurs, since the update probability across all keys is assumed to be uniform in the first place.

The IN-PLACE update queue behaves exactly the way described above. It is guaranteed that the first enqueue time of any update key in the IN-PLACE queue is definitely earlier than all the others behind that key in the queue. Therefore, once a key gets to the head of the queue, it is the one with the earliest first enqueue time as well as with the largest queue staleness across all keys. As a result, the IN-PLACE key scheduling policy ensures that the overall average staleness (application as well as queue) is minimized for a uniform distribution of update key frequencies.

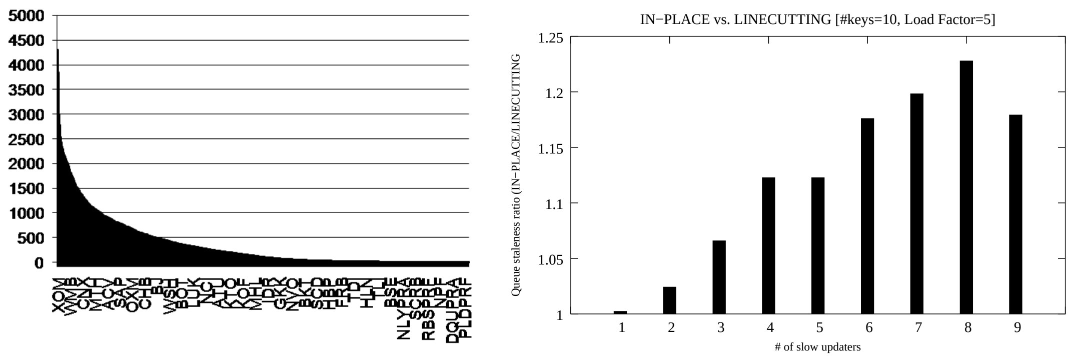
5.3 Linecutting

The previous section showed that IN-PLACE key scheduling policy can not be beaten if all keys update at the same frequency. However, often times the update frequencies are not uniform. Figure 10(a) illustrates this situation for financial market data taken from NYSE Trade & Quote (TAQ) database for a trading day in January 2006 [2]. More than 3000 different stock symbols (i.e., update keys) were involved, and the most active of these updated for 4305 times whereas the least active one updated for only once during the corresponding day. Is IN-PLACE key scheduling policy still the best that we can do for such non-uniform update frequencies? If not, how can we exploit the differences in update frequencies to find a better scheduling algorithm?

The idea behind our linecutting key scheduling algorithm (LINECUTTING) is based on the following two observations:

- Non-uniform key update frequencies: Normally, IN-PLACE treats all keys the same so that all of them wait for the same amount of time in the queue. This time is proportional to the queue length multiplied by the cost for processing a tuple. Therefore, by speeding up occasional residents of the queue by letting them to the front of the queue allows for a queue wait time per key to be changed in correlation to its update rate.
- Bursty loads: Slow updaters may update all at once or at least in a very close time vicinity of each other. Between such bursts, staleness for such keys will remain zero, and whenever a burst happens, the system must switch focus into updating the application with the slow ones, by allowing the slow updaters to cut into the front of the queue.

The key point here is to be able to immediately identify the slow updaters when they arrive. Slow updaters are the keys that have such slow update frequencies that between two updates, all other more faster updaters that are in the queue have time to be scheduled at least once. Formally, if u_k is the update rate of a slow updater, then the following inequality should hold: $u_k < 1/c * n$, where c is the cost for processing one tuple, and n is the total number of residents in the queue at the time when k updated. Our LINECUTTING algorithm works by testing this inequality for each new arrival; in case of success, the key will be allowed to cut into the front of the queue. Otherwise, the IN-PLACE update policy applies.



(a) NYSE TAQ data (daily # of updates per stock symbol

(b) Linecutting

Figure 10: Non-uniform update frequencies

To test the effectiveness of our algorithm in minimizing staleness, we considered a non-uniform distribution of update keys containing two categories of updaters: very slow and fast. For example, given 10 keys, out of which, 9 keys update 10 times more than the 10th key would be such a distribution. For testing this heuristic, we have varied both the load factor and the number of slow updaters in the distribution. Figure 10(b) shows our result for when load factor was 5. On the x-axis, we vary the number of slow updaters in our 10-key dataset, and on the y-axis, we show queue staleness ratio between IN-PLACE and LINECUTTING. When we have no/all slow updaters (i.e., x=0/x=10), both algorithms perform exactly the same (i.e., y=1). As we increase the number of slow updaters, we start seeing the benefits of the LINECUTTING heuristic. At 8 slow updaters, the have the highest benefit, and then we see some drop. This is due to the fact that too many slow updaters start satisfying the linecutting condition, and therefore the queue starts behaving like an IN-PLACE queue. Note that we repeated this experiment for other load levels as well, and obtained a similar result. In conclusion, under non-uniform key update rates, sufficiently slow updaters to cut into the front of the queue decreases overall average staleness for all update keys.

6 Handling Windows

In this section, we focus on window management in UpStream. Window manager is concerned with two issues: (i) ensuring the correct update execution semantics for sliding window-based query processing, (ii) managing the memory window buffers accordingly.

In what follows, we will focus on the logical aspects of our solution for the above issues. The physical implementation aspects will be presented in the next section.

6.1 Correctness Principles

Sliding window-based queries take as input a window that consists of a finite collection of consecutive tuples and apply an aggregate processing on this window, which results in a single output tuple for that window. Unlike the tuple-based case where the unit of update is a single tuple both at the input and at the output, for the window-based queries, the unit of update is a window for the input and a single tuple for the output. Due to this difference in operational unit, our problem has an additional challenge compared to the tuple-based scenario: Besides worrying about the recency of our outputs, we must also make sure that that they correspond to the results of “semantically valid” input windows.

We define semantic validity based on the “subset-based” output model used by previous work on approximate query processing and load shedding (e.g., [26]). This model dictates that if we can not deliver the full query result for some reason (e.g., overload), then we are allowed to miss some of the output tuples, but the ones that we deliver must correspond to output values that would also be in the result otherwise. This requires that the original window boundaries are respected and the integrity of the kept windows are preserved.

Based on the above, we adopt the following two design principles for handling windows correctly in UpStream:

- All or nothing: Windows should either be processed in full, or not at all.
- No undo: If we decide to open a window and start processing it, we must finish it all the way to the end. Changing decisions on a partially processed window is not allowed. In this case, we say that we “committed” to that window.

These two principles help us produce correct outputs. Of course we would still need to determine which windows we should commit to, but this decision would affect staleness, not correctness. We come back to this issue in Section 6.3.

6.2 Window-Aware Update Queues

We satisfy the above correctness principles using the tuple-marking scheme that was introduced in our previous work on load shedding for aggregation queries [26]. In this work, load shedding is achieved through a special Window Drop operator which injects window keep/ drop decisions into the input tuples by marking them with window specification bits. These marks are then interpreted by the downstream aggregate operators, which can be arranged in arbitrary compositions (pipeline, fan-out, or their combinations). As a result, subset results are produced.

In UpStream, we have essentially pushed the above tuple marking logic down to the storage level, by making our update queue "window-aware". Instead of a window drop operator, the UpStream update queue manager marks the tuples inside the storage before the query processing on those tuples actually start. As described in [26], a tuple has a special field called win_spec which bears the window marks. One marks a tuple with a zero or positive value only if it is a window starter. In UpStream, the update queue and the operator must work together. The update queue keeps track of the most recent window update. When it is time to open a new window for processing, the queue marks it with a non-zero window specification corresponding to the expected closing timestamp. Older windows are marked with 0. On the other side, the operator will open windows selectively based on the window specification marks. Table 1 gives the possible actions that the operator might do upon reading a tuple from the queue, based on the win_spec values.

Table 1: Relevant operator actions

t	**win_spec**	**relevant action**
win start	$\tau > 0$	open window $W[t, \tau]$
		where τ is the expected window end,
		update state
	0	do not open window, update state
	-1	update state
$t \geq W.end$	don't care	prepare to emit result for W

Our window-aware update queues have one additional advantage over the window drop approach: windows which are redundant (if any) can be identified and their tuples can be immediately pruned inside the update queue before they hit the query processor (this is analogous to the in-place updates in the tuple-based case). Our approach also differs in how we decide on which windows to mark for dropping. The window drop approach does this in a probabilistic way, by setting a drop probability to be applied on a batch of windows. In other words, which windows are dropped does not matter as long as the drop probability is set high enough to remove the excess load in the system so that query latency is kept under control. In UpStream, for lowering staleness, we must keep the most recent windows, therefore, the update queue marks the windows accordingly.

Next, we want to define what is the most recent window update ($MRWU$). For tuple-based updates, each new arrival (t, x, val_t) constitutes the most recent update for key x and overwrites the previous unprocessed one $(t-1, x, val_{t-1})$. For the windowing case, the $MRWU$ rule considers updates in terms of windows. Assuming ordered arrivals for the same key, a window ω is represented by a pair $[start, end]$, where $\omega.start$ represents the first tuple in the window (window starter), and $\omega.end$ is the last tuple in the window (window closer).

Now, we are ready to define the $MRWU$ rule. Consider that a tuple (t, x, val_t) for key x has just been enqueued. The most recent window update rule is stated by defining an order relation on windows. This can be done in two ways:

1. $MRWU_{end}$: a window ω_i is more recent than a window ω_{i-1} if both are fully arrived (all the window tuples have timestamps less than t) and $\omega_{i-1}.end < \omega_i.end \leq t$. Basically, using this rule, the operator would commit to the most recent window that has fully arrived.

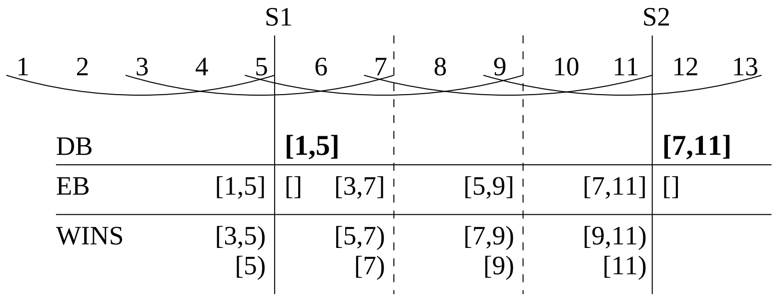
2. $MRWU_{start}$: a window ω_i is more recent than a window ω_{i-1} if ω_i is partially arrived and $\omega_{i-1}.start < \omega_i.start \leq t$. In this case, the operator would commit eagerly to the most recently started window even though it hasn't closed so far.

Based on these rules, there are two variant implementations of choosing which windows to commit to in UpStream, which lead to two different window buffer management approaches, as we explain next.

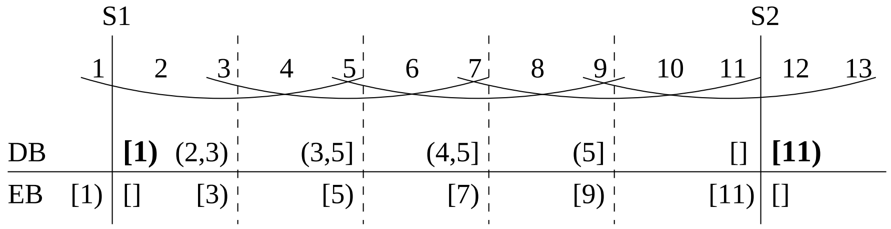
6.3 Window Buffer Management

In UpStream, window buffers maintain the data structures for keeping track of window updates. Since updates of different key values are handled independently, there is one window buffer for each update key value. Let's focus on one of these buffers. It consists of two main data structures: Enqueue Buffer (EB) and Dequeue Buffer (DB). EB holds information regarding the most recently arrived window (either partial or full) and DB holds information regarding the last "committed" window (i.e., its processing has started but not necessarily ended). Please note that EB and DB do not contain actual tuples, but only the intervals indicating window boundaries.

Over the course of system execution, the window manager maintains these data structures so that our correctness principles are met as well as update semantics is imposed. Two main factors affect how this maintenance is done: (i) scheduling frequency which depends on the query processing cost and the key scheduling policy (in case of multiple update keys), and (ii) when we commit to a window, for which there are two major alternatives: at window ends vs. at window starts. The first factor is beyond the control of the window manager. However, the second factor can be directly controlled by implementing the window buffers accordingly. For this, we have come up with two alternative window buffer management policies based on



(a) Lazy Window Buffer



(b) Eager Window Buffer

Figure 11: Window Buffer Management Policies

whether the commit decision is taken at the window ends (the “lazy buffer”) or at the window starts (the “eager buffer”).

6.3.1 Lazy Window Buffer

The lazy approach is a direct adaptation of the tuple-based update processing approach: the query only consumes a fully arrived window at each scheduling time point, and it must be the most recently arrived one. We call this approach lazy, since the window commit decision is postponed until we are sure that we have a full window.

Figure 11(a) shows a trace of the lazy window buffer in action for a sliding window with size $w = 5$ and slide $s = 2$. Assume that the integers correspond to tuple timestamps which determine the window boundaries. $[a, b]$ means that a and b are the boundaries for a fully arrived window. $[a, b)$ means that the window has started at a and its elements until b have arrived. Solid vertical lines marked with S1 and S2 show the scheduling points, and vertical dashed lines indicate *window ends*. The bold intervals show the full windows that are committed to by the lazy policy.

In order to keep track of all the incomplete windows, the lazy buffer maintains an additional list (WINS). The top of the WINS list contains the oldest started window (e.g., $[3, 5)$). When this window closes (e.g., $[3, 7]$), it is popped out of the WINS and placed in the EB. If the EB is not empty at this time, we say that the window that is already in the EB is overwritten. This is what happens with windows $[3, 7]$ and $[5, 9]$ in our example. At the scheduling points, the DB will pop the interval in the EB (if any), which represents the most recently arrived full window (e.g., $[1, 5]$ and $[7, 11]$). Those will exactly be the windows that the buffer manager commits to.

6.3.2 Eager Window Buffer

The eager approach can be seen as an adaptation of append-based windowing to update-based semantics. As in the append case, we commit to windows from their starting points (no necessarily to all the starting windows). We only commit to the latest started ones at each scheduling time point. We call this approach eager, since the window commit decision is eagerly taken as soon as a fresh window is seen (even if it has not fully arrived yet).

Figure 11(b) shows a trace of the eager window buffer in action for a sliding window with size $w = 5$ and slide $s = 2$. Again assume that the integers correspond to tuple timestamps which determine the window boundaries. $[a)$ means that a marks the boundary of a recently started window. (a, b) means that elements of a previously committed window between the boundaries a and b are available for dequeue. Both $(a, b]$ and $(a]$ have similar meanings, except that they also show that the closing element of a previously committed window is also available for dequeue. As before, solid vertical lines marked with S1 and S2 show the scheduling points, but different from before, vertical dashed lines indicate *window starts*. Finally, the bold intervals show the recently started windows that are committed to by the eager policy.

The eager buffer does not need an additional WINS list for incomplete windows, as it already keeps track of all the incomplete windows that matter for its operation. The EB holds the most recently started window. Window updates happen on window starters. The EB is always emptied when a new window starter arrives (e.g., $[3)$, $[5)$, $[7)$, $[9)$, $[11)$). At the scheduling points, the DB will pop the interval in the EB (if any), which represents the most recently arrived partial window (e.g., $[1)$ and $[11)$). Those will exactly be the windows that the buffer manager commits to.

7 UpStream Implementation

In this section, in order to put all the pieces together, we show how the UpStream Update Queue Manager implements the techniques that we have presented in the previous sections.

As shown in Figure 12, the UpStream Update Queue Manager consists of three main layers: Key Scheduler (KS), Window Manager (WM), and Memory Manager (MM). KS manages staleness by deciding the order of update key to be processed next; WM manages the window buffers as described in the previous section; and MM takes care of physically storing and garbage collecting the actual data tuples in the memory.

7.1 Key Scheduler

KS contains a Key Hash Table (KHT) that maps each distinct key value to its corresponding key cell. Key cells are linked together to form a queue of keys, while the “head” refers to the key cell that is to be dequeued next. Each key cell holds a pointer to the lower-level data structures for a key. In the tuple-based processing case, we only need to hold a direct pointer to the memory location where the latest arrival is stored. In the window-based processing case, each key cell has a separate window buffer which is managed by the WM.

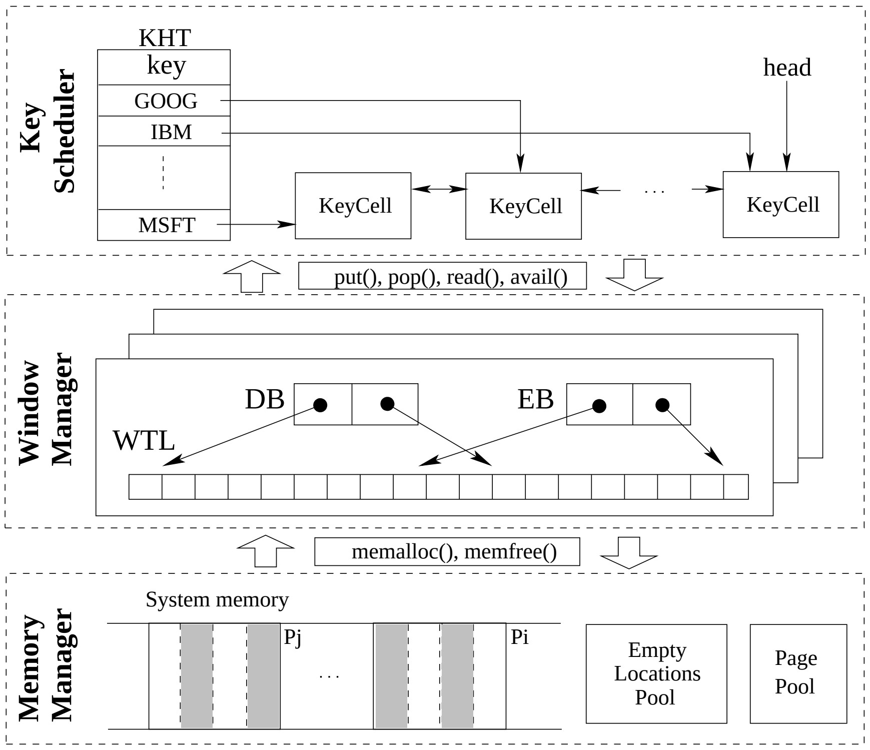


Figure 12: Update Queue Manager Implementation

Event	Key status	Action
INS	Not in the update queue and/or under processing	Insert into update queue
UPD	In the update queue or under processing	Rearrange/Set pending

Table 2: Key Scheduler events.

For a better understanding of update queues and how they behave in a stream processing system, we have identified the main stages that a key can go through in the system. The existence of a key, hence its life-cycle, is given by the fact that it has been observed on the input stream, it is being processed or waiting to be processed. The life-cycle of a key is depicted in the upper part of Figure 13 with the basic states and possible transitions.

Table 3 describes the states of a key life-cycle together with the main events related to it. In the lower part of Figure 13, one can find an example of how the state of key 'a' evolves given a series of updates in the input stream. Updates are depicted on the bottom timeline while the results of processing the key are shown on the top one. The second timeline shows the events issued by the Key Scheduler. Each key update triggers either an INS or an UPD event. These events are explained in Table 2. The Key Scheduler places the key in the update queue according to the key scheduling strategy. For instance, IN-PLACE puts the key at the back of the queue at INS events or leaves it in the same position for UPD events (rearrange does

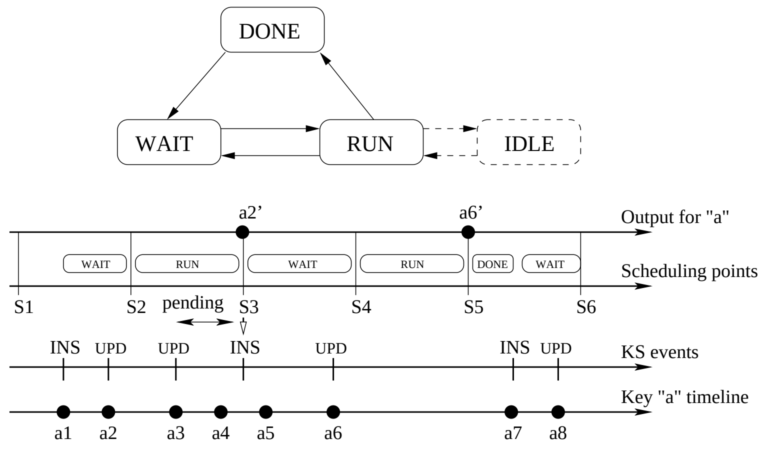


Figure 13: Key life-cycle and main events.

nothing). To complete the picture, a delete event (which always refers to the head of the queue) originates in the system Scheduler and represents the moment when the query starts processing the key. We will denote such events as scheduling points S_i. (see the third timeline).

7.2 Window Manager

In UpStream, windowing over update streams is implemented as window buffers, one for each key. The management of window buffers (WM) is orthogonal to the management of keys (KS). The KS communicates with the window buffer in the WM through a typical sequential storage interface (pop(), put(), read(), avail()). While we discuss the implementation of the storage interface, we want to point out the main issues that a window buffer has to address.

Holding window tuples

As we have seen in Section , the $MRWU$ rule gives us the design principles for building the window buffers. The data structures and their role in the eager and lazy windowing buffers have already been explained in Sections 6.3.1 and 6.3.2. Inside a window buffer, we have the DB and EB data structures that hold references to the committed window and the most recent window update, respectively. DB and EB represent window structures (either full or partial) and do not hold the actual tuples. These are stored sequentially and without redundancy by using a Window Tuple List (WTL) which in turn holds pointers to the actual memory locations where the window tuples are physically stored. This list is indexed by unique identifiers given to each tuple. The ordering of unique identifiers is the same as the tuple-arrival order. We represent windows similarly to intervals. Thus, window structures are merely holding information about the beginning and the end of the window using the tuple identifiers. An important property is that one can iteratate through the tuples of a window simply by incrementing between the beginning of the window and the end.

State	**Description**
DONE	This state symbolizes the fact that a key has no pending updates and that the application has the latest results for it. A key reaches this state with 0 staleness.
WAIT	A key enters this state when the Key Scheduler issues an INS event which effectively inserts the key into the update queue. While in the WAIT state, further updates for the same key observed on the input stream will mark an UPD event, leaving the key state unchanged. The only effect is to update the storage associated to the key with the new value(s) (e.g. given an IN-PLACE policy). Scheduling points cause the transition from WAIT to RUN state.
RUN	A scheduling point brings the key which sits at the head of the update queue into RUN state where it stays until the query has finished processing the associated key value(s). While in this state, a key may have updates in the input stream. We must avoid the Key Scheduler issuing an INS event and putting the key in the WAIT state at the same time it is in the RUN state. To this extent, we say that the key is *pending* and the Key Scheduler only issues an UPD event. When the query is done with the key, two things can happen: (i) either the key is placed into WAIT state if it was pending (e.g.: S_3) or (ii) the key makes the transition to DONE if there are no newer updates for it (e.g.: S_5).
IDLE	A key can be placed in this state only if it has previously been in the RUN state. The reason could be related to the fact that the query has to wait for further key values to arrive. For example, for multi-key update streams, it may be the case that a window for one key has gaps due to arrivals for other keys. For such cases, the eager windowing semantics imposes the use of the IDLE state. A key moves to this state when the query hits a gap in the committed window. The key moves back to the RUN state when further tuples belonging to the committed window arrive. While some keys are IDLE, others are in RUN state. This means that the EAGER windowing semantics makes full use of the CPU time, despite the gaps. To facilitate keys going back and forth between IDLE and RUN states, we have introduced the $promote$ function in the window buffer interface. This function is called upon key arrivals to test whether the key needs to be placed directly in the RUN state without having to go through the WAIT state. Since the IDLE state has no meaning for the LAZY windowing semantics, $promote$ always returns false in this case.

Table 3: Key life-cycle states.

Besided EB and DB, the window buffers maintain state of their own. For instance, we have already talked about the WINS list used by the lazy window buffer. The eager window buffer does not need to keep track of additional window structures. However, it keeps state symbolizing the most recent window starter (denoted by recent_start) and the expected committed window end (denoted by keep_until). recent_start always points to the first tuple in EB, whereas keep_until would point to the last tuple in DB (after the whole committed window has arrived).

Algorithm 1 Eager Window Buffer $put()$.

```
Require: tuple $t$ to enqueue
Ensure: $t$ is stored and window buffer updated
 1: $location[t] \leftarrow$ Memory-Manager-Alloc()
 2: $utid[t] \leftarrow$ Get-Next-Uid()
 3: Insert(WTL, $utid[t]$, $location[t]$)
 4: Store($t$)
 5: if $t$ is STARTER then
 6:     $recent\_start \leftarrow win\_val[t]$
 7:     $mark[t] \leftarrow 0$ {mark with DISALLOW by default}
 8:     $expired\_tuples \leftarrow$ EWB-Safe-Remove-Zones-EB(WTL) {remove tuples from WTL to garbage collect}
 9:     for all tuple $et$ in $expired\_tuples$ do
10:         Memory-Manager-Free($et$) {effectively calls the MemoryManager's memfree() method}
11:     end for
12: end if
13: if $win\_val[t] \leq keep\_until$ then
14:     Add-Last(DB, $utid[t]$)
15: end if
16: if $win\_val[t] \geq recent\_start$ then
17:     Add-Last(EB, $utid[t]$)
18: end if
```

Garbage collection

Window update buffers perform overwriting of older windows with newer ones. Overwriting entails that some or all of the overwritten window tuples expire from the storage. Normally, the contents of EB are subject to garbage collection. However, some or all of the EB tuples can also belong to the committed window or to the newly forming window(s). Our techniques guarantee that these tuples do not accidentally expire in the process by identifying the zones in EB that are safe to get rid of. These zones consist of window tuples with consecutive unique identifiers.

Window buffer access

However different in operation the eager and lazy window buffers are, they share some features. For instance, element access (read()) and availability check (avail()) are the same:

→ avail() is called for the key in RUN state to check whether the window buffer has more window tuples. Basically, if DB containing the committed window has tuples, avail() returns true.

→ Provided that DB is not empty, read() always returns the first tuple in DB. This holds for both types of window buffers, since the query always works with tuples from a committed window.

Algorithm 2 Lazy Window Buffer $put()$.

```
Require: tuple $t$ to enqueue
Ensure: $t$ is stored and window buffer updated
 1: $location[t] \leftarrow$ Memory-Manager-Alloc()
 2: $utid[t] \leftarrow$ Get-Next-Uid()
 3: Insert(WTL, $utid[t]$, $location[t]$)
 4: Store($t$)
 5: if $t$ is STARTER then
 6:     $W \leftarrow$ New-Window($utid[t]$) {A window structure containing only this tuple}
 7:     Add-Last(WINS, $W$)
 8:     $mark[t] \leftarrow 0$ {mark with DISALLOW by default}
 9: else
10:     $mark[t] \leftarrow -1$ {don't care}
11: end if
12: if $t$ is CLOSER then
13:     $expired\_tuples \leftarrow$ LWB-Safe-Remove-EB(WTL) {remove tuples from WTL to garbage collect}
14:     for all tuple $et$ in $expired\_tuples$ do
15:         Memory-Manager-Free($et$) {effectively calls the MemoryManager's memfree() method}
16:     end for
17:     EB $\leftarrow$ Pop-Front(WINS)
18: end if
19: for all window $W$ in WINS do
20:     Add-Last($W$, $utid[t]$)
21: end for
```

Window buffer modifiers

Adding and consuming tuples to and from window buffers is handled by put() and pop(). When adding a window tuple, put() performs several operations (see listings 1 and 2):

$\rightarrow$ *Account for the enqueued tuple*: after computing a unique identifier for the tuple and getting an available memory location from the Memory Manager (by calling memalloc()), the tuple is physically stored in memory and the location is registered at the end of WTL with the assigned identifier.

$\rightarrow$ *Account for tuple relevance*: this is where the two window buffers start to behave differently. In the eager window buffer case, the relevant tuples are the ones that start windows. A new window starter triggers an overwrite operation resulting in the new tuple replacing the existing tuples in EB. For the lazy window buffer, both window starter and closers are relevant. Each window starter adds a corresponding window structure to the end of the WINS list, whereas each window closer takes out the top of the WINS list and replace the contents of EB with it, triggering an overwrite of the window that was there before. Overwriting of the EB buffer is done safely using the EWB-Safe-Remove-EB() and LWB-Safe-Remove-EB() methods for eager and lazy window buffers, respectively. The former ensures that the intersection between DB (in case it contains anything) and EB does not expire. The same must happen for the latter, where in turn, the intersection between EB and the top of the WINS list must also be kept.

→ *Update window structures*: for sliding windows, a tuple can belong to more windows (as many as the overlap factor o). Since window structures only keep information about the beginning and the end of the window, whether it is full or partial, the window structures that include the enqueued tuple must be updated: their end moved to include the new tuple identifier. Eager window buffer updates DB, if the tuple is within the limits of the committed window (i.e. not greater than keep_until). EB is updated only if recent_start is not the one corresponding to the committed window. The lazy window buffer always updates all the window structures in the WINS list.

When the query consumes a window tuple, it is the first tuple in DB. For this, pop() has to take the following actions (see listings 3 and 4):

Algorithm 3 Eager Window Buffer $pop()$.

```
Require: DB not empty or EB not empty
Ensure: return a window tuple and free up its underlying storage
 1: if keep_until is unset then
 2:     DB ← EB
 3:     Clear(EB) {effectively unset the EB window structure}
 4: end if
 5: $to\_deq$ ← Get-First(DB)
 6: if $win\_val[to\_deq]$ = recent_start and keep_until unset then
 7:     keep_until ← Compute-EWE($to\_deq$) {compute the expected window end for the window started by
        $to\_deq$}
 8:     unset recent_start
 9:     $mark[to\_deq]$ ← keep_until
10: end if
11: if $win\_val[to\_deq]$ = keep_until then
12:     unset keep_until
13: end if
    {safely remove the tuple from WTL}
14: if EWB-Safe-Remove(WTL, $to\_deq$) then
15:     Memory-Manager-Free($to\_deq$) {effectively calls the MemoryManager's memfree() method}
16: end if
17: return $to\_deq$
```

→ *Commit and mark*: the window buffer must realize whether it has to commit to the window sitting in EB. Lazy window buffer commits if and only if DB is empty. Eager window buffer in turn commits if DB is empty and keep_until is unset. Committing means DB becomes EB and the latter window structure is unset. The immediate effect is to mark the first tuple in DB with the expected window end (the keep_until for the eager window buffer is also set to this value).

→ *Garbage collection*: the dequeued tuple can belong to other windows besides the committed window. Garbage collecting this tuple must be handled using the same principles as in the case of window overwrite, but applied to one tuple. Regardless of this, DB must always move its beginning to the next tuple in the window immediately after the presently dequeued tuple.

Algorithm 4 Lazy Window Buffer $pop()$.

```
Require: DB not empty or EB not empty
Ensure: return a window tuple and free up its underlying storage
 1: if DB not empty then
 2:     DB ← EB
 3:     Clear(EB) {effectively unset the EB window structure}
 4:     $do\_mark$ ← true
 5: end if
 6: $to\_deq$ ← Get-First(DB)
 7: if $do\_mark$ = true then
 8:     $mark[to\_deq]$ = Compute-EWE($to\_deq$) {compute the expected window end for the window started
        by $to\_deq$}
 9: end if
    {safely remove the tuple from WTL}
10: if LWB-Safe-Remove(WTL, $to\_deq$) then
11:     Memory-Manager-Free($to\_deq$) {effectively calls the MemoryManager's memfree() method}
12: end if
13: return $to\_deq$
```

7.3 Memory Manager

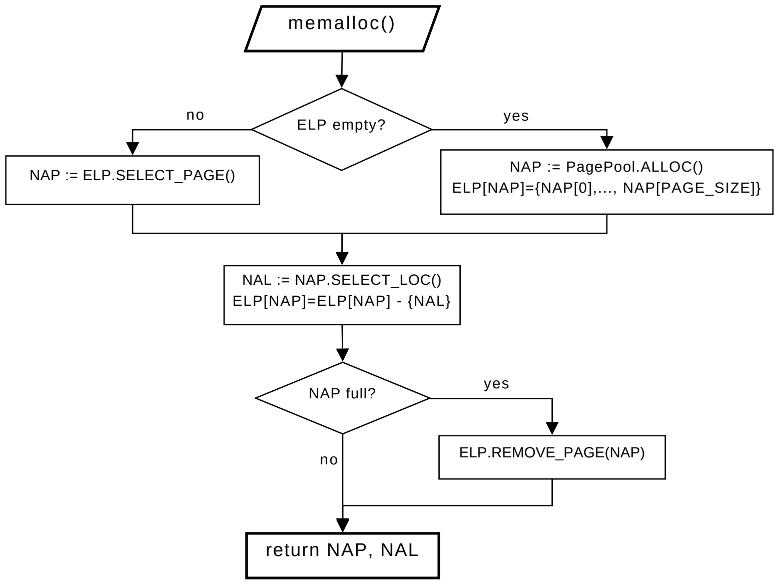
Finally, the physical memory locations are handled by the MM layer. The WM communicates with the MM with memalloc() and memfree() methods, which in turn makes sure that new window tuples are allocated and expired window tuples are garbage collected. In MM, a Page Pool contains a number of pages allocated from the system memory. To avoid memory proliferation, we keep a Pool of Empty Locations (ELP) indexed by pages.

Figure 14 shows the operations behind memalloc() and memfree(), which we explain next:

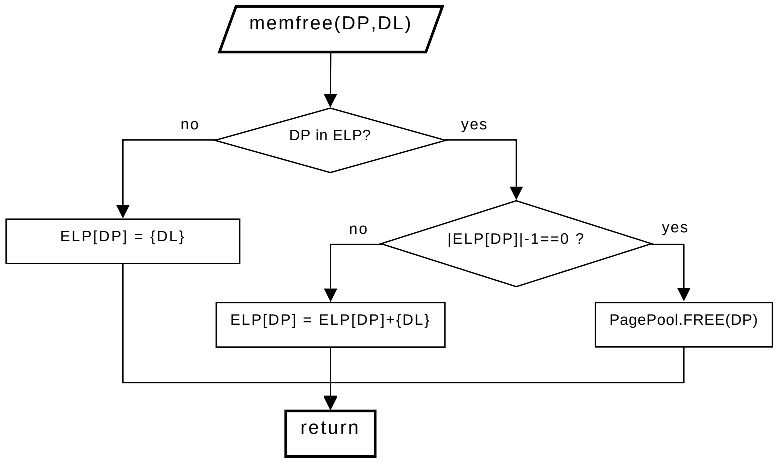
→ memalloc() (Figure 14(a)) is called at enqueue only if a new memory location is needed. Basically, it computes the next available location for storing a tuple, which is given by a page (NAP) and a location offset within that page (NAL). Memory locations are properly sized in order to fit tuples according to the stream schema. The mechanism tries to retrieve a page from ELP, which holds the pages with available locations. If it turns out that ELP is empty, this means either all pages have been garbage collected, or all pages that are in use are full. Hence, the mechanism will ask the PagePool to allocate a new page. Once the NAP page is located, the first available location within it is set as NAL.

→ memfree() (Figure 14(b)) is called at each dequeue with the location of the dequeued tuple, given by a page (DP) and an offset within that page (DL). If DP was full before this dequeue, it is added to ELP together with the dequeued location. Otherwise, we update the list of available locations for DP by adding DL to it. If it turns out that DP just became empty, the page is freed (given back to the PagePool).

The general policy of our memory management is to eagerly consume the locations in ELP and lazily free pages only when they become empty. This way, space utilization is localized and memory proliferation avoided.



(a) On-demand paging with pooling.



(b) Garbage collection.

Figure 14: Memory Manager operations.

8 Performance

In this section, we provide an experimental evaluation of the storage-based load management techniques proposed by UpStream. The main goals of our experimental study are as follows:

- to show how our storage-based load management approaches that use in-place update queues compare against the state of the art load management approaches that use append queues, in terms of both output staleness as well as memory consumption.
- to evaluate how effectively we are able to manage load on both tuple-based as well as sliding window-based queries.

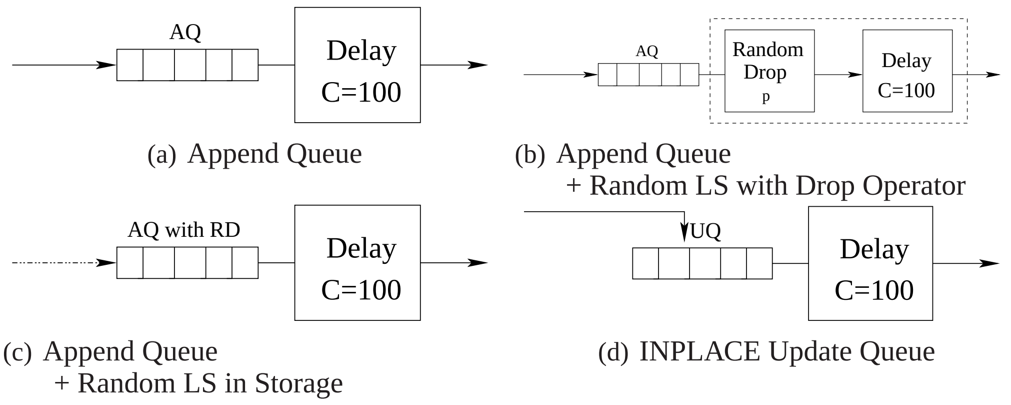


Figure 15: Tuple-based Processing Scenarios

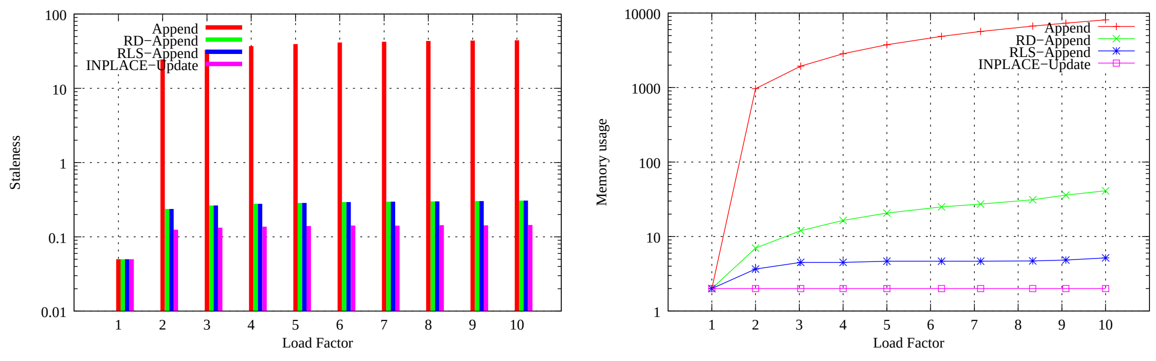
- to show how our techniques scale with: (i) increasing input load in terms of both arrival rates and number of update keys, (ii) changing query properties such as window size, slide, and the relative placement of the sliding window operator in a given query, and (iii) different window processing modes (incremental vs. non-incremental processing).

Next, we will first describe our experimental setup, then we will present our results under two query scenarios: tuple-based and window-based processing.

8.1 Experimental Setup

System: We implemented our UpStream framework as part of the Borealis stream processing system, expanding on its storage manager component [3]. The QoS and statistics monitoring components of Borealis were also extended in order to compute staleness and memory usage. We configured the Borealis scheduler to operate on a query-at-a-time basis (i.e., the "super-box" scheduling technique [11] for processing latest updates through the query as a whole). The motivation was to minimize the per-key processing overhead due to context switches. In all of our experiments, we have used a single-node setup for Borealis running on a Linux box with an Intel Quad Core Intel Xeon 3360 2.8GHz processor and 8GB of memory.
Workload: In all experiments, we used synthetically generated tuples of the form `(time, update key, value)`. The input is ordered by `time`. The number of distinct `update key` values ranges between 1 and 100, depending on the experiment. The actual values of the `value` field does not have any significance in terms of what we measure in the experiments, thus, they were randomly chosen from a numeric domain. The input rates were set according to the desired level of overload to be exerted on the system, given a query workload with a specific estimated processing cost per tuple.

On this data, we have defined two main classes of queries: tuple-based queries and sliding window-based queries. In each experiment, we use a single query, whose operators are scheduled "in one go" with the input tuples available for processing at that scheduling step. In order to be able to control the processing cost of this query, we use the "delay" operator of Borealis.



(a) Staleness vs. Load factor (b) Memory usage vs. Load factor

Figure 16: Update Queue vs. Append Queue Variants (single update key)

A delay operator simply withholds its input tuple for a specific amount of time (busy-waiting the CPU) before releasing it to its successor operator. A delay operator is essentially a convenient way to represent a query plan with a certain CPU cost; its delay parameter provides a knob to easily adjust the query cost. In the experiments, we set this delay parameter to either 100 msec or 50 msec. In the tuple-based scenarios, using one delay operator was sufficient for the purpose, whereas in window-based scenarios, we have also used a sliding window aggregate operator that is placed either upstream or downstream from a delay operator. This helps us explore if the position of the sliding window operator relative to the rest of the query has any effect on performance. We will further describe the details of these query scenarios when we present the corresponding results for them in the following sections.

Performance Metrics: We have primarily used two performance metrics in our experiments:

- Staleness: As formulated in Section 3.3, first, per-key average staleness is computed over individual staleness values of all output tuples generated for each distinct update key value across a given run period, and then, an overall average is computed across all the keys. Staleness is measured in seconds.
- Memory usage: Maximum memory allocated for the UpStream page pool is recorded between consecutive output deliveries. Then we compute a maximum over all these recordings to see the worst-case memory usage for UpStream across a given run. Memory usage is measured in the number of stored tuples.

Note that each of our experiments has been repeated 10 to 20 times and the performance values we report are averages across these repeated runs. The purpose was to eliminate the side effects of: (i) certain probabilistic choices in our scenarios (such as arrival times and orders of different update keys in multi-key scenarios, drop probability in random drop scenarios); (ii) the randomness in the order of certain system events that is happening beyond our control (such as how the enqueue thread and the scheduler thread in Borealis are synchronized by the operating system).

8.2 Results on Tuple-based Processing

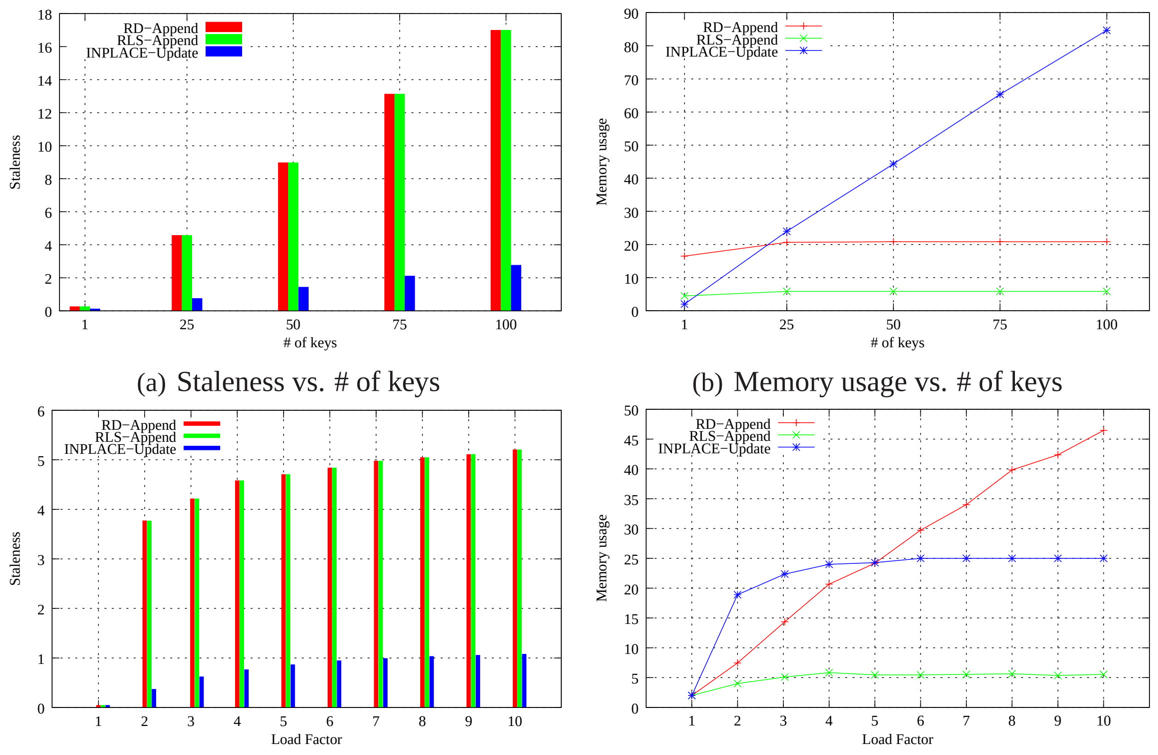
In this experiment, we use a tuple-based query Q with CPU cost of $C = 100$ msec per tuple, and compare the following scenarios:

- Append Queue (Figure 15(a)): Input to Q is fed by a traditional append queue with FIFO ordering (i.e., no load is shed).
- Append Queue + Random Load Shedding with Drop Operator (Figure 15(b)): Input to Q is fed by a traditional append queue and a Random Drop operator with a proper drop probability p is inserted to the query plan in order to shed the excess load [25].
- Append Queue + Random Load Shedding in Storage (Figure 15(c)): Input to Q is fed by a traditional append queue which is extended with the ability to apply random drops inside the queue to shed the excess load (i.e., the queue plays the role of the Random Drop operator in the previous scenario).
- INPLACE Update Queue (Figure 15(d)): Input to Q is fed by UpStream's INPLACE update queue.

8.2.1 Single Update Key

First, we ran the four scenarios in Figure 15 with an input stream consisting of a single update key value. We measured staleness and memory usage for increasing degrees of load in the system. Our results are shown in Figure 16. Note that, the Load Factor (LF) on the x-axes shows the ratio between the query's time cost and the time between consecutive tuple arrivals. Thus, the system is running at the capacity level at $LF = 1$, and is overloaded when $LF > 1$.

In Figure 16(a), at $LF = 1$, all scenarios produce the same average staleness, since there is no tuple accumulation in the queues and therefore no need for load shedding. In this case, the update queue also behaves exactly like an append queue (i.e., no inplace updates). However, when the load factor goes beyond 1, append queue can not keep up with the input arrival rates any more, and starts accumulating tuples. This causes staleness to dramatically increase up to unacceptably high levels (note the log-scale on the y-axis). The situation is not as bad for the rest of the scenarios, although we see that INPLACE update queue scenario scales much better with increasing load than the random load shedding-based scenarios. This is due to the fact that an update queue always ensures that the latest (i.e., most fresh) tuple is kept in the queue, whereas in random load shedding, although queue growth is avoided, it is not guaranteed that the queue will always keep the most fresh tuple since the tuple drop decisions are made probabilistically. An interesting observation was also that the difference between doing load shedding at the storage level or at the query processing level via a drop operator turned out to be small. This shows that the major role in higher staleness levels is played by the randomness of the drop decision, and not so much by where the decision is made. Finally, we should also note that, an additional advantage of the update queue over load shedding is that it naturally adapts to increasing load without the need to tune any knobs in the system, whereas in random load shedding, the system has to properly set the drop probability p according to the current load level.



(a) Staleness vs. # of keys (b) Memory usage vs. # of keys

(c) Staleness vs. Load factor (d) Memory usage vs. Load factor

Figure 17: Update Queue vs. Random Load Shedding Variants (multiple update keys)

In Figure 16(b), we again see that the memory usage for the append queue dramatically increases as the load increases due to continuous tuple accumulation in the queue. The situation looks much better for the rest of the scenarios. As expected, in the storage-centric approaches (INPLACE update queue and random load shedding in storage), the memory usage is essentially constant and lower. The reason is that the queue immediately responds to a newly arriving tuple (keep it or drop it), avoiding the need to redundantly store tuples until the query consumes them. The difference in memory usage for the drop-based load shedding scenario is because a higher number of tuples have to wait in the queue until the drop operator decides which ones will be dropped. This result clearly shows that managing load at the storage level helps reduce memory consumption.

8.2.2 Multiple Update Keys

Next, we ran three of the scenarios in Figure 15 ((b)-(d)) with an input stream consisting of multiple update key values [5]. We measured staleness and memory usage for increasing number of update keys as well as increasing degrees of load in the system. Our results are shown in

[5]Scenario (a) was omitted since single-key experiments already showed how badly append queue behaved under overload.

Figure 17.

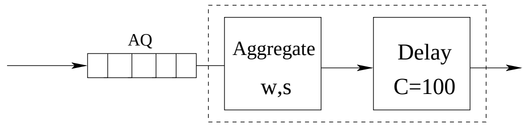
First, we varied the number of keys between 1 to 100 (x-axis), while fixing the load factor at $LF = 4$. In each run, keys were set to update with the same average frequency. For instance, for 10 keys and 1000 arrivals, each key would be observed about 100 times. On the other hand, time between consecutive arrivals of the same key was not necessarily even (for a given run as well as across repeated runs), since we have used a ZipF distribution with skew parameter $\theta = 0$ [16] and with a different seed value at every repeated run. The results of this experiment are shown in Figures 17(a) and 17(b). All three compared approaches show increasing staleness with increasing number of keys. Staleness grows for the INPLACE update queue scenario, because the average length of the update queue grows when more keys update uniformly at the same time. In this case, all keys wait in the queue for nearly the same amount of minimum possible time, which is directly correlated with the number of distinct update keys. The load shedding scenarios however, do not exhibit the same type of correlation. The reason for this is that random load shedding approaches are not key-aware, i.e., drop decisions are made completely randomly across different key values. The effect of this on staleness becomes even more apparent as the number of keys increases. The trend for memory usage on the other hand is somewhat the opposite. As in the single-key experiment, load shedding approaches consume fixed memory space, though storage-based variant is slightly more efficient than the operator-based variant. However, update queue's memory usage is directly proportional to the number of distinct update keys, since the queue maintains about one slot per key at steady state when all keys are updating at uniform frequency.

Second, we varied the load factor between 1 and 10, while setting the number of keys to 25. The results of this experiment are shown in Figures 17(c) and 17(d). Both results agree with the results of the single-key scenario presented in Section 8.2.1, with one major difference: The memory usage of our update queue is correlated with the number of update keys (in this case, 25), which requires more memory slots to be allocated.

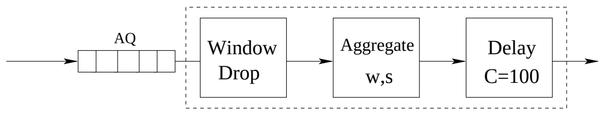
8.3 Results on Window-based Processing

In this experiment, we use variants of a window-based query Q, with a sliding window aggregate operator with window size w and slide s, and a delay operator with cost C, and compare the following:

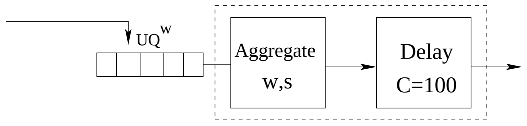
- Append Queue (Figure 18(a)): Input to Q is fed by a traditional append queue with FIFO ordering (i.e., no load is shed).
- Append Queue + Random Load Shedding with Window Drop Operator (Figure 18(b)): Input to Q is fed by a traditional append queue and a Window Drop operator with a proper drop probability p and batch size B is inserted to the query plan in order to shed the excess load [26].
- INPLACE Update Queue + Upstream Windowing (Figure 18(c)): Input to Q is fed by UpStream's window-aware INPLACE update queue. This scenario represents a "Cost Per-Window" (CPW) case, where the window is constructed and its result is produced before the rest of the query operations are applied on that result. In this scenario, we also assume



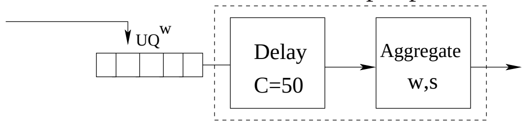
(a) Append Queue



(b) Append Queue + Random LS with Window Drop Operator

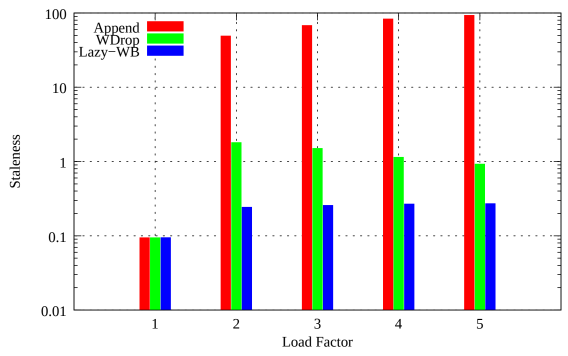


(c) INPLACE Update Queue + Upstream Windowing (CPW)

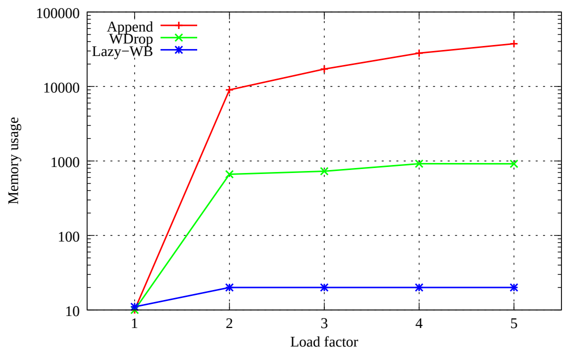


(d) INPLACE Update Queue + Downstream Windowing (CPT)

Figure 18: Window-based Processing Scenarios



(a) Staleness vs. Load factor



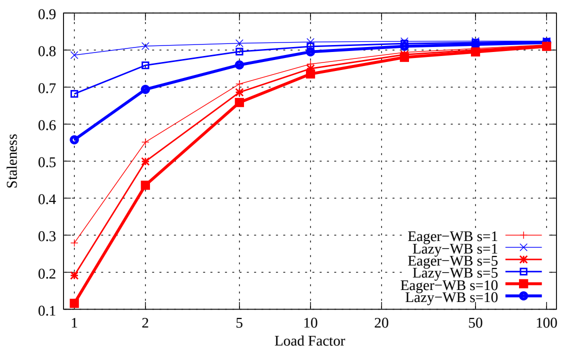
(b) Memory usage vs. Load factor

Figure 19: Window-Aware Update Queue vs. Append Queue Variants (CPW, LAZY-WB, single update key)

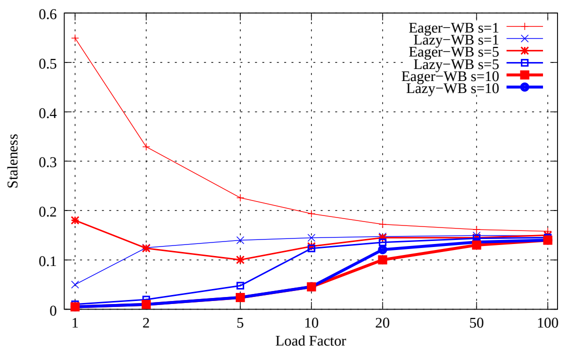
a "non-incremental" window processing mode, in which the aggregate function is applied when the aggregate operator has the full window (i.e., all window tuples are consumed as a batch rather than individually).

- INPLACE Update Queue + Downstream Windowing (Figure 18(d)): Again, the input to Q is fed by UpStream's window-aware INPLACE update queue, with one major difference in the query plan: the order of the aggregate and delay operators is switched. As such, this scenario represents a "Cost Per-Tuple" (CPT) case, where the window is constructed and its result is produced after the rest of the query operations are first applied on the input. In this scenario, we also assume an "incremental" window processing mode, in which the aggregate function is applied incrementally as each tuple of a given window arrives.

In the next set of experiments, we first show the benefit of using a window-aware update queue in terms of both staleness and memory usage over the state of the art. Then we focus on the window buffer management aspect of our window-aware update queues, and present an in-depth evaluation of how the eager and lazy techniques perform under different parameter

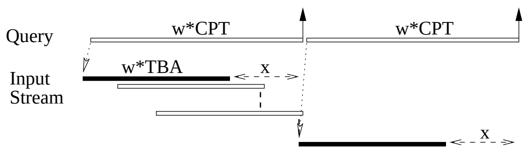


(a) CPT on Delay + Windowing

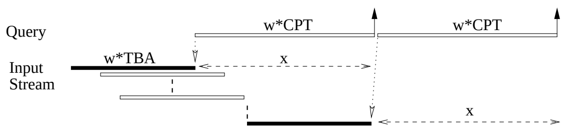


(b) CPW on Windowing + Delay

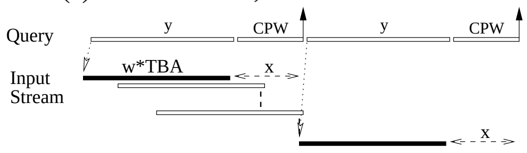
Figure 20: Staleness for Eager vs. Lazy Window Buffer Management on Window-Aware Update Queues (single update key)



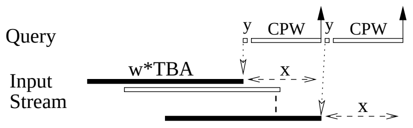
(a) EAGER-WB, CPT



(b) LAZY-WB, CPT



(c) EAGER-WB, CPW



(d) LAZY-WB, CPW

Figure 21: Window Buffer Behavior for CPT and CPW (TBA = Time Between Updates, CPT = Cost Per Tuple, CPW = Cost Per Window)

settings, first in terms of staleness, and then memory usage.

8.3.1 Using a Window-Aware Update Queue

First, we ran scenarios (a)-(c) in Figure 18 with an input stream consisting of a single update key value. We set the window buffer management to LAZY-WB and window processing mode to CPW. The reason for these settings was to create a similar load management scenario as in the window drop-based scenario to make the results more directly comparable. In that scenario, window drop decisions are made for the whole window. With the lazy approach, UpStream also behaves in a similar way. In this experiment, we also used a simple tumbling window with $w = s = 10$ and set window drop's batch size $B = 10$. Essentially, these settings create an ideal scenario for the window drop to be the most effective. This way, we can see how much better we can perform (if any), even in such an ideal setting for the state of the art.

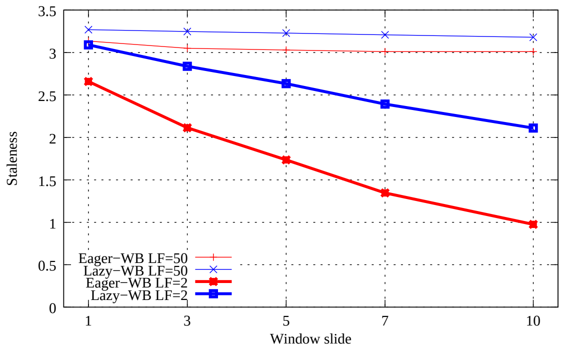
Our results are shown in Figure 19. As in the case of tuple-based processing, we can clearly see that, if no measure is taken, staleness can go through the roof with increasing load illustrated by the behavior of the append queue. Using a window drop operator certainly relieves the problem by dropping batches of windows, and hence, at least keeping the queue sizes and tuple latencies under control. However, again the drop decision is made without considering the recency of the windows. Therefore, as in the tuple-based processing case, staleness is still higher compared to our window-aware update queue. An additional interesting result we see on the graph of Figure 19(a) is that, for window drop, the staleness seems to slightly decrease as the load grows. This is due to the fact that at excessive load levels, the drop probability gets higher, gradually reducing the need for making the right choice about keeping the more recent windows. Finally, on the memory usage front, the result that we depict in Figure 19(b) verifies the result that we obtained for the tuple-based processing scenario shown earlier in Figure 16(b).

8.3.2 Window Buffer Management: Eager vs. Lazy

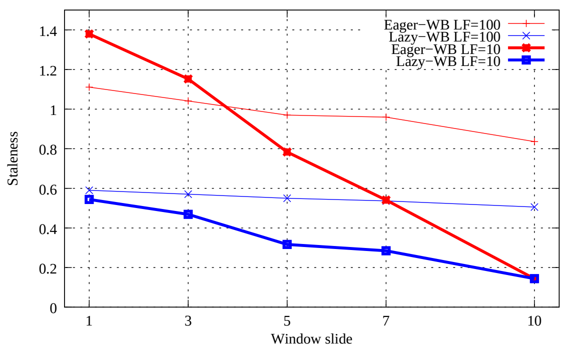
In this section, we focus on comparing the eager and lazy window buffer management strategies for our update queues.

Single Update Key:
For this experiment, we ran the scenarios in Figure 18(c) and 18(d) with EAGER-WB or LAZY-WB respectively, and measured how their staleness changed with increasing load. The window size w is set to 10, but the window slide s is varied as 1, 5, and 10.

The results are shown in Figure 20. We present two graphs: the one in Figure 20(a) focuses on CPT-style processing on the scenario of Figure 18(d), and the one in Figure 20(b) focuses on CPW-style processing on the scenario of Figure 18(c). Overall, we see that in both graphs, staleness seems to stabilize with increasing load, which shows how our window-aware update queues in general scale with load. Of course, the window slide value and the window buffer management strategy both create variations in staleness behavior, which we will further explain below. However, it is important to note at this stage that, by looking at these two graphs side-by-side, we can immediately observe that EAGER-WB performs better than LAZY-WB when the delay operator is in front of the windowing operator (CPT), and vice-versa when the delay is applied after the windowing operator (CPW). Therefore, we will explain our results around a detailed analysis of the CPT and CPW scenarios.
The CPT Case: In Figure 20(a), staleness achieved by EAGER-WB and LAZY-WB stabilizes towards the same level with increasing load. We will further explain this matter using Figure 21, which shows how EAGER-WB (Figure 21(a)) and LAZY-WB (Figure 21(b)) behave in time in the CPT scenario. At the top of the figures, we depicted the time spent by the query to produce an output after committing to one window. At the bottom, one can see window formation over the input stream. A window takes $w * TBA$ time units to arrive, while to be processed by the query, it takes $w*CPT$. x denotes the time elapsed between the closing of the committed window (depicted as a thick line) and the delivery time for that window, and it forms the basis for the staleness growth. LAZY-WB does not let the query start processing until it has a fully-formed window. This yields $x = w * CPT$. In the EAGER-WB case, x has a wider



(a) CPT

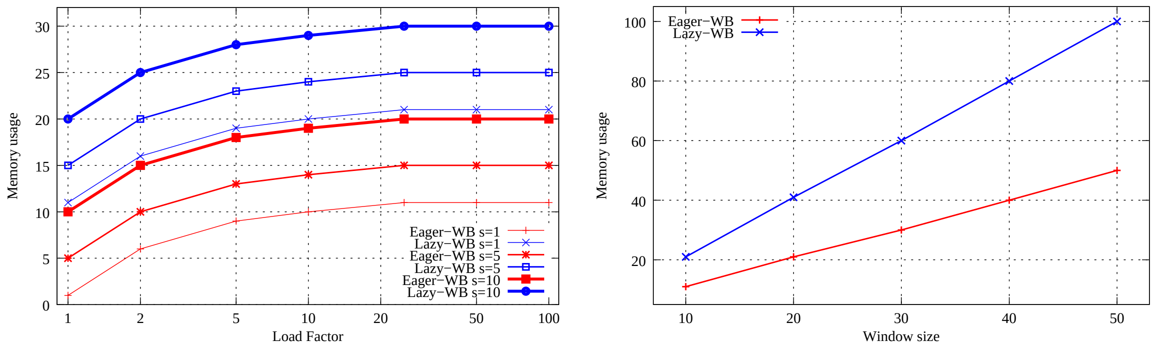


(b) CPW

Figure 22: Staleness for Eager vs. Lazy Window Buffer Management on Window-Aware Update Queues (multiple update keys)

range for variation. The general formula would be $x = w * CPT - (w-1) * TBA$. The lower bound for x is found in the normal load scenario ($CPT = TBA$) and is equal to CPT. That is, after a window has closed, the query needs another CPT time units to update the window state and deliver the result. When load increases beyond the system capacity, x grows. We can rewrite the formula by considering the load factor: $x = TBA + w * CPT * (1 - \frac{1}{LF})$. For very high LF values, the upper bound for x tends to approach to $w * CPT$, which is the value achieved by LAZY-WB. This is confirmed by the results shown in Figure 20(a), where we can see that for the first half of the x-axis, EAGER-WB achieves lower staleness than LAZY-WB.
The CPW Case: To explain the result of Figure 20(b), we have depicted the behavior of the EAGER-WB and LAZY-WB in Figures 21(c) and 21(d). In the CPW scenario, a window result spends CPW time units until it is delivered to the output stream. This means that $x = CPW$ for both window buffers. Despite this, the results of our experiment show that EAGER-WB behaves worst than LAZY-WB for the first half of the considered load spectrum. In the second part, where the load is very high, the two techniques stabilize staleness around the same level. This can be explained if we consider y, the time spent by the aggregate operator to dequeue and compute the aggregated result. For LAZY-WB, y is negligible compared to the cost of the downstream query (CPW), since the window is already in the storage and the operator can consume it very fast. In the case of EAGER-WB on the other hand, the aggregate operator dequeues and computes window tuples as they arrive. In this case, y can get to the maximum of $w * TBA = w * \frac{CPW}{LF}$. This is only reached in the non-overload scenario, explaining the strange maximum that is achieved by EAGER-WB at $LF = 1$. As LF increases, y becomes smaller and smaller, making EAGER-WB approach LAZY-WB for high load levels.

Multiple Update Keys:
Next, we repeated the previous experiment for multiple number of update keys. For this experiment, we set the number of update keys to 10 and the window size as $w = 10$, and varied the window slide between 1 and 10. Smaller window slide means higher load as it corresponds to slowly sliding windows that highly overlap and lead to more number of windows to be con-



(a) Memory usage vs. Load factor (b) Memory usage vs. Window size

Figure 23: Memory Usage for Eager vs. Lazy Window Buffer Management on Window-Aware Update Queues

structed and processed. We took measurements for EAGER-WB and LAZY-WB for two load levels, low ($LF = 2$ or $LF = 10$) and high ($LF = 50$ or $LF = 100$).

Figure 22 shows our results, again with a separate graph for CPT and CPW. In both graphs, we observe a similar trend: staleness decreases almost linearly with increasing window slide. This is natural, since a greater window slide incurs less load. A special situation we expect to observe in the multi-key scenario is that windows of a given key may experience time gaps due to tuple arrivals for the other keys. We expect LAZY-WB not to exhibit any sensitivity to such gaps since it only processes fully-formed windows. EAGER-WB, on the other hand, can be sensitive to gaps. We will further explain the results of this experiments through a detailed analysis around the CPT and CPW scenarios.

The CPT Case: In Figure 22(a), at very high load, EAGER-WB and LAZY-WB are closer in terms of staleness. We have also measured the average length of the queue for the entire runtime, and for $LF = 50$, the reported values for both EAGER-WB and LAZY-WB indicated that, on average, the queue contained all the keys. Despite this, EAGER-WB seems to be doing better than LAZY-WB. The reason is that it takes advantage of gaps in the windows by immediately starting to serve the other keys. x equals to $k * CPT$ where k is the number of keys that have closed their committed windows. For high load, k is very small and staleness is mostly affected by the number of keys in the queue. LAZY-WB achieves the same value for x as in the single-key scenario ($w * CPT$) and hence staleness is affected by the number of keys in all cases. For lower load levels, k is less than w, yielding smaller staleness for EAGER-WB.

The CPW Case: In Figure 22(b), we see that EAGER-WB starts to be affected by gaps in a rather negative way. If we place Figure 21(c) in the current context, the only thing that changes is y. This is where gaps come into play. y is the time spent by the aggregate to dequeue and compute a window for a key. The problem now is that a window takes longer to arrive, hence y will also include the accumulated length of the gaps.

Memory Usage:

In this last experiment, we would like to contrast the memory requirements of our two window buffer management strategies. We only include our results on the CPT-style processing, as we observed that it had more influence on memory usage than CPW.

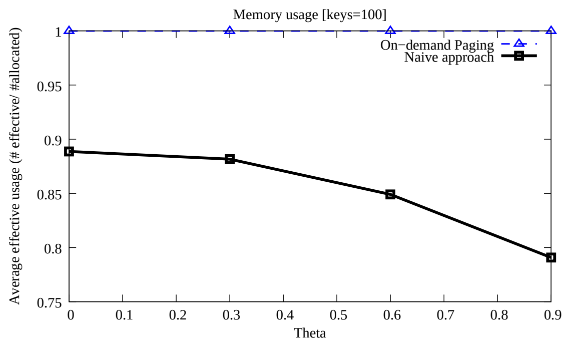
We expect EAGER-WB and LAZY-WB to have a difference in memory usage due to the difference in the way windows are overwritten. We expect that the amount of consumed memory would also depend on the window size, window slide, and the system load.

Our first graph (Figure 23(a)) shows memory usage for different load factors and window slide values. The theoretical worst case bounds for EAGER-WB and LAZY-WB approaches are $w + s$ and $2 * w + s$, respectively. We can see from the graph, that these bounds are reached at very high system loads, where memory usage stabilizes. Our techniques achieve bounded memory usage.

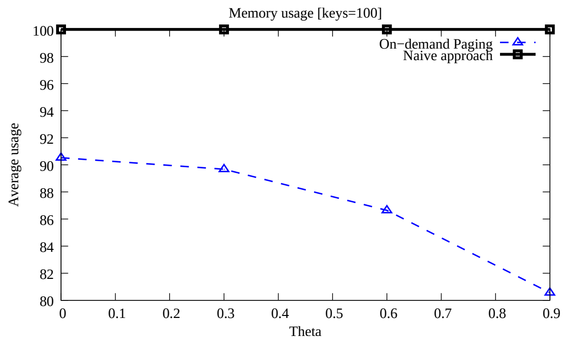
Our second graph (Figure 23(b)) shows memory usage for different window sizes. The load factor was set to $LF = 50$ and the window slide was set to $s = 1$ (i.e., high load, extreme degree of window overlap). It is clearly seen that the memory requirements for both buffering techniques grow linearly with window size, which shows that UpStream does not add much overhead beyond the normal expected requirements. This experiment also shows once again that EAGER-WB requires less memory than LAZY-WB.

8.4 Evaluation on memory management

In this section, we focus on the behavior of the our memory management mechanism. To keep things simple, this evaluation was conducted for the tuple-based scenario with multi-key update streams where a key needs only one memory location to store its latest update. Nevertheless, the conclusions of this evaluation can be applied to the window-based scenario as well, since we have seen in Section 8.3.2 that the window buffers have bounded memory requirements.



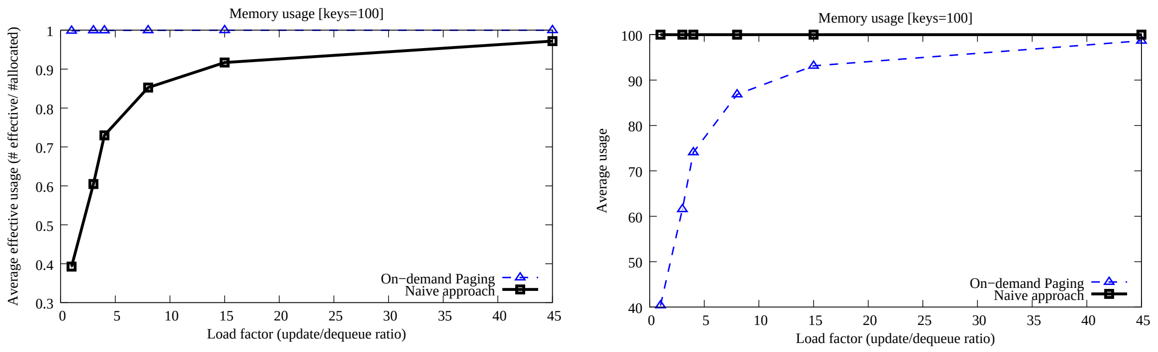
(a) Effective memory usage



(b) Memory usage

Figure 24: Various update rate distributions

In order to have a comparison term, we have developed a naive implementation of memory management for update streams in which a Key Cell always keeps its memory location. The Naive Manager never performs garbage collection. Our experiments consisted of the following setup:



(a) Effective memory usage (b) Memory usage

Figure 25: Various load factors

- we used several datasets with uniform and skewed distributions of 100 keys. In order to vary the skewdness between key update rates we used the Zipf distribution. The θ parameter of our Zipf generator had values in the interval [0,1). In our experiments, we used the following values for the θ parameter: 0 (uniform distribution), 0.3, 0.6 and 0.9 (very high skew).
- we sent tuples into the system at increasing rates between 20 and 1000 tuples per second while keeping the processing rate (or dequeue rate) constant at 10 per second.
- the metrics that we used for comparison were:
  - *average usage*: how many memory locations were used on average. The average usage is computed as an average across runs with different distributions.
  - *average effective usage*: the percentage of the total allocated memory that was actually in use on average (non-dirty).

We measured average usage and average effective usage while varying the θ parameter (see Figure 24) and the load factor (see Figure 25). The on-demand paging mechanism performs better in terms of effective usage of allocated space. When we increase the load in the system, this means that more keys update between two consecutive dequeues. Increasing skew among the key update rates means that the queue is smaller on average than in the uniform distribution case. Figures 24(a) and 25(a) show that the on-demand paging mechanism maximizes the usage of the allocated space. A value of 1 means that no matter the distribution, this mechanism does not introduce dirty locations (allocated by not used). On the other hand, while increasing skewdness, the naive approach uses the allocated space (in the order of number of keys) less efficiently since there is no need to keep locations for keys that are updating slowly. While varying the load factor, the effective usage for the naive approach is low for small load factors but it improves for higher load. If we look at the average usage (Figures 24(b) and 25(b)), the naive approach keeps the locations for all the keys blocked, whereas on-demand paging eagerly tries to release freed locations to the PagePool.

We have seen that a naive memory management mechanism needs a constant level of memory usage which turns out to be the maximum needed for an update stream. However, depending on the parameters of the update stream and the load, it may not be useful to have the maximum space allocated at all times, which makes our Memory Manager with on-demand paging and eager garbage collection a better fit.

9 Conclusions and Future Work

In this report, we have argued that we need new load management techniques for streaming applications with update semantics, since these applications care more about staleness than latency. We proposed a novel storage-centric load management framework based on update queues. We further devised a detailed analysis and a set of new techniques for update key scheduling, for space-efficient window processing techniques for ensuring correct and low-staleness results for sliding window queries. We would like to address the following issues as part of our future work:

- We have shown that IN-PLACE update queue policy minimizes staleness when update keys are uniformly distributed in terms of their frequencies. Our ongoing work has evidence that better key scheduling algorithms can be devised for the non-uniform case. Therefore, our first future work item is to extend our UpStream framework to include additional key scheduling policies that may be a better fit for other workload types.
- In this report, we focused on minimizing staleness for a single continuous query on streaming data with multiple update keys. We would like to extend our techniques to scheduling multiple continuous queries, possibly with sharing.
- Currently, we assume continuous access frequencies across all update keys at the end-point application. However, the application may also want to access the results at different rates (e.g., would like GOOG stocks to refresh every 1 minutes, while GM stocks to refresh every 1 hour). Therefore, we intend to integrate application access frequencies into our QoS model.
- Lastly, our storage framework allows append and update queues to co-exist in the system at the same time. We will explore how we can optimize our storage framework under such scenarios. One interesting idea is to investigate adaptive schemes that allow the system to automatically switch between the two queuing modes based on changing load.

References

[1] Intelligent Transportation Systems (ITS). `http://www.its.dot.gov/`.

[2] NYSE Data Solutions. `http://www.nyxdata.com/nysedata/`.

[3] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, January 2005.

[4] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying Update Streams in a Soft Real-Time Database System. In *ACM SIGMOD Conference*, San Jose, CA, June 1995.

[5] B. Adelberg, B. Kao, and H. Garcia-Molina. Database Support for Efficiently Maintaining Derived Data. In *EDBT Conference*, Avignon, France, March 1996.

[6] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, 15(3), September 1990.

[7] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE Conference*, Boston, MA, March 2004.

[8] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *NSDI Conference*, San Fransisco, CA, March 2004.

[9] M. H. Bateni, L. Golab, M. T. Hajiaghayi, and H. Karloff. Scheduling to Minimize Staleness and Stretch in Real-Time Data Warehouses. In *ACM SPAA Conference*, Calgary, Canada, August 2009.

[10] I. Botan, G. Alonso, P. M. Fischer, D. Kossmann, and N.Tatbul. Flexible and Scalable Storage Management for Data-intensive Stream Processing. In *EDBT Conference*, Saint Petersburg, Russia, March 2009.

[11] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[12] J. Cho and H. Garcia-Molina. Synchronizing a Database to Improve Freshness. In *ACM SIGMOD Conference*, Dallas, TX, May 2000.

[13] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream Warehousing with DataDepot. In *ACM SIGMOD Conference*, Providence, RI, June 2009.

[14] L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling Updates in a Real-Time Stream Warehouse. In *ICDE Conference*, Shanghai, China, March 2009.

[15] L. Golab and T. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2), June 2003.

[16] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *ACM SIGMOD Conference*, Minneapolis, MN, May 1994.

[17] B. Kao, K. yiu Lam, B. Adelberg, R. Cheng, and T. S. H. Lee. Updates and View Maintenance in Soft Real-Time Database Systems. In *CIKM Conference*, Kansas City, MO, November 1999.

[18] C. Olston and J. Widom. Best-Effort Cache Synchronization with Source Cooperation. In *ACM SIGMOD Conference*, Madison, WI, June 2002.

[19] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *IEEE ICDE Conference*, Atlanta, GA, April 2006.

[20] H. Qu and A. Labrinidis. Preference-Aware Query and Update Scheduling in Web-Databases. In *IEEE ICDE Conference*, Istanbul, Turkey, April 2007.

[21] H. Qu, A. Labrinidis, and D. Mosse. UNIT: User-centric Transaction Management in Web-Database Systems. In *IEEE ICDE Conference*, Atlanta, GA, April 2006.

[22] F. Reiss and J. M. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *IEEE ICDE Conference*, Tokyo, Japan, April 2005.

[23] M. A. Sharaf, A. Labrinidis, P. K. Chrysanthis, and K. Pruhs. Freshness-Aware Scheduling of Continuous Queries in the Dynamic Web. In *WebDB Workshop*, Baltimore, MD, June 2005.

[24] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *VLDB Conference*, Vienna, Austria, September 2007.

[25] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[26] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *VLDB Conference*, Seoul, Korea, September 2006.

[27] Y. Tu, S. Liu, S. Prabhakar, and B. Yao. Load Shedding in Stream Databases: A Control-Based Approach. In *VLDB Conference*, Seoul, Korea, September 2006.

[28] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing Resiliency to Load Variations in Distributed Stream Processing. In *VLDB Conference*, Seoul, Korea, September 2006.

[29] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *IEEE ICDE Conference*, Tokyo, Japan, April 2005.