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 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 **1**

2 Related Work **3**

3 Problem Context **4**
3.1 Data Model 4
3.2 Query Processing Model 5
3.3 Quality of Service Model 7
3.4 Problem Definition 9

4 Storage-centric Approach **9**
4.1 Update Queues 10
4.2 Storage Manager Architecture 11

5 Minimizing Staleness **11**
5.1 Application Staleness vs. Queue Staleness 11
5.2 FIFO and IN-PLACE Update Queues 14
5.3 Linecutting 16

6 Windowing **18**
6.1 Assumptions 19
6.1.1 Windowing queries 19
6.1.2 QoS model revised 19
6.1.3 Window drops 20
6.2 Windowing techniques 21
6.2.1 LAZY windowing semantics 23
6.2.2 EAGER windowing semantics 25
6.3 Evaluation 27
6.3.1 Windowing semantics 27
6.3.2 Multiple keys and arrival patterns 28

7 Memory Management **29**
7.1 Paging Mechanism 32
7.2 Window Manager 33
7.3 Evaluation 36

8 Conclusions and Future Work **39**

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 to adaptive load shedding. 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 data 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 situations 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. Similarly, in a facility monitoring system, one might like to watch for the *latest* 5-minute average temperature in all rooms. In such applications with update semantics, the primary 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 possible *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 stale-

ness. For streams with append semantics, various load shedding techniques have been proposed to deal with the latency issue. 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 missing is a framework that ensures that this is done in a systematic way to directly minimize staleness.

UpStream defines a *a storage-centric framework* to 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, while efficiently handling large numbers of update keys with different update frequencies. In addition to theoretical analysis results, we also present experimental evidence showing the effectiveness of our algorithms via a practical simulator program as well as via an implementation on top of a data stream management system prototype using both synthetic as well as real datasets.

To summarize, UpStream makes the following contributions:

- a new update-based data stream processing model,
- a storage-centric load management framework for update streams based on update queues,
- techniques for minimizing application staleness for non-uniform update key distributions and sliding window queries, and
- theoretical and experimental performance analysis.

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 assumptions that provide the context for the research problem that we are addressing. Section 4 introduces our storage-centric load management framework which divides the problem into three orthogonal components, which we then address in the following sections: Section 5 focuses on the problem of minimizing staleness by frequency-sensitive update key scheduling; Section 7.2 describes how we can extend these techniques to queries with sliding windows in a way to ensure correctness and low-staleness, while using the memory efficiently. A detailed description of the underlying memory management mechanisms is provided in Section 7. Our

theoretical and experimental analysis results are presented alongside with the corresponding techniques that we propose. Finally, we conclude with a discussion of future work directions in Section 8.

2 Related Work

We summarize related work in three categories. First, we compare our work against the directly related work in stream load management. Then we look at two other non-streaming contexts, web database synchronization and materialized view maintenance, where reducing staleness for updates has also been a concern.

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., [21]) as well as dynamically changing this placement as data arrival rates change (e.g., [6], [22], [12]). 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., [18], [5], [15], [20], [17]). 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 [8]. 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 [14, 13]. The former must meet timeliness requirements based on a deadline, while the latter must meet freshness requirements based on a threshold defined in terms of the number of unapplied updates. A user satisfaction metric is defined that combines the two, which needs to be maximized using an adaptive load control scheme that performs admission control for query

transactions and update frequency modulation for update transactions. The update transactions in this line of work have a very similar semantics to our update streams. The major difference of our work is due to 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 [16]. 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, done within the context of real-time database systems in particular, has 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). Furthermore, read transactions on both the base data as well as materialized views must also be executed, with specific deadlines (view export). As in web databases, this separation creates a tradeoff between response time for read transactions and freshness for update transactions. Adelberg et al proposes scheduling algorithms for efficient updates on base data [2], as well as on derived data [3]. The focus of the former work is on transaction priority policies taking advantage of different importance values for different data objects as well as arrival rate differences for update requests vs. read requests. The focus of the latter work is on exploiting update locality by batching related updates into a single recomputation on the view so as to deal with the high cost of applying recomputations on derived views. Kao et al further extend these works by proposing scheduling policies for ensuring absolute and relative temporal consistency to increase the number transactions that meet their deadlines [10].

3 Problem Context

In this section, we describe the basic models and assumptions that underlie our work and define the research problem we are addressing.

3.1 Data Model

We model data streams as totally ordered sequences of tuples where each tuple represents an update to the one before it in the sequence. Tuples in a stream are

ordered in ascending order of a time field. Furthermore, one or more fields in the stream schema can be designated as the *update key*. In this case, a tuple updates a previous tuple with the same key value. For example, given a stream of stock prices in the form of (time, 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). If no update key is specified, then each tuple in the stream is simply an update of the one before it.

To generalize, an *update stream* consists of a sequence of relational tuples with the generic schema (time, 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. These two assumptions are important and have certain implications on query operator behavior, which we will describe next as part of our query processing model.

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 basic continuous query models. Figure 1 illustrates these alternatives. Unless otherwise stated, we will focus on the push-push model in this report.

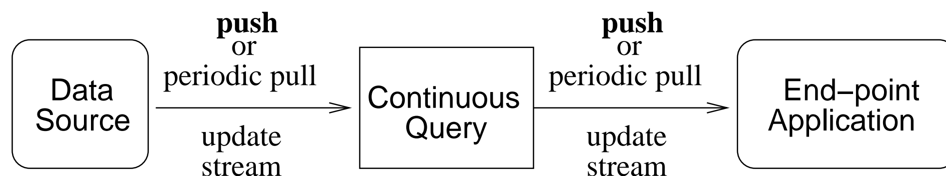


Figure 1: Continuous query models

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 and newer results must be produced as soon as possible. Otherwise, the application may perform actions based on a value which is in fact stale (i.e., superseded by a newer one) 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. This introduces a new query processing model with a slightly different correctness criteria. 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. In other words, the application cares more about freshness than completeness. This is a radically different semantics than that in the case of append-based applications.

Continuous queries are composed of a number of stream-oriented operators. In this work, we focus on five such operators that are commonly used in streaming applications [9]. In what follows, we briefly describe these operators, with a special emphasis on how they should behave under the new update-based stream data model.

- *Filter:* Filter applies a predicate on each input tuple and drops the ones that do not satisfy this predicate. Its behavior on update streams is the same as it is on append streams.
- *Map:* Map transforms an input tuple into an output tuple by applying a function on it. In case of update streams, the function is not allowed to modify the update key fields, since the key must be retained in the output tuple.
- *Aggregate:* Aggregate groups an input stream into substreams by their group-by fields and constructs sliding windows on each substream based on a given window size and a window slide. 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, since an aggregate operates on a per-window basis, it essentially maps 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 7.2.
- *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

[1] In UpStream, we simply assume that the group-by fields are the same as the update key fields. We will address the more general superset case as part of our future work.

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 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*, in order to capture this goal.

Staleness metric was also used by previous work for bringing multiple data

[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 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.

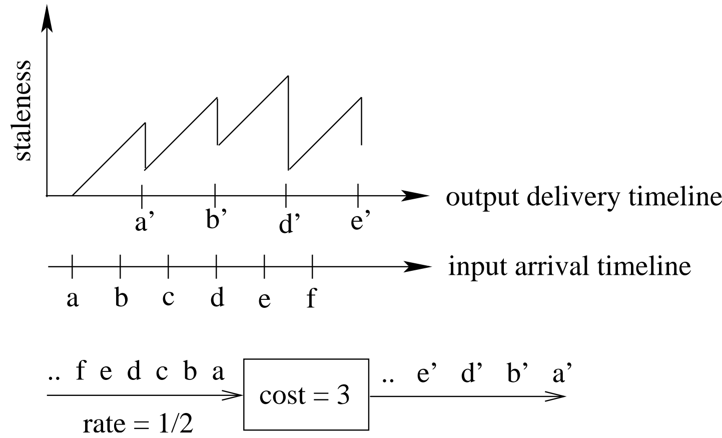


Figure 2: Staleness

copies up-to-date in various different contexts including cache synchronization and materialized view maintenance [4, 8, 11, 10]. 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, 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 continuous 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 continuous queries in the system during a given system run time period.

3.4 Problem Definition

Based on the models we introduced above, we can define our 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.

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 [7].

In the rest of this section, we will first introduce the concept of update queues; then we describe the design of our storage manager architecture to support the implementation and optimization of different types of queues in the system.

4.1 Update Queues

Traditional append-based query processing semantics requires tuple queues be modeled as append-only data structures with FIFO semantics. To support the new update-based query processing semantics, we have extended this 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 arrival; 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 three 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 3 shows the architecture of our storage manager. Storage manager interfaces with input sources, output applications, and query processor through its iterators. These iterators enable 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 (e.g., key update frequencies) to drive its optimization decisions. The underlying queue semantic can either be a traditional append queue or an update queue. In UpStream, we focus on the latter.

The update queue manager is broken into two main components. The key scheduler decides when to schedule different update keys for execution. As we will explain shortly in the next section, the key scheduler can choose to apply one

[3]For ease of presentation, we are simply assuming tuple-based queries here; extensions to window-based queries will be provided later in Section 7.2.

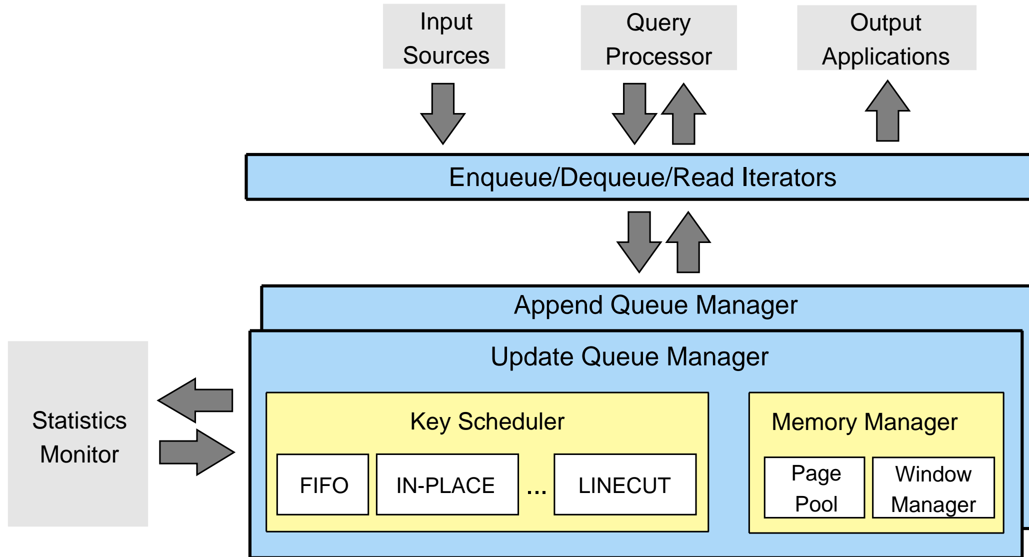


Figure 3: UpStream storage manager architecture

of several policies including FIFO, INPLACE, and LINECUTTING. The memory manager component oversees page allocation, and for sliding window queries, also takes care of window management. In our implementation, we handle key scheduling, page pool management, and window semantic management as three orthogonal issues. However, page pool management and window management logically belong together as part of the memory manager component since they both focus on space efficiency, while key scheduling is mainly concerned with staleness.

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 by service differentiation across different update keys.

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.

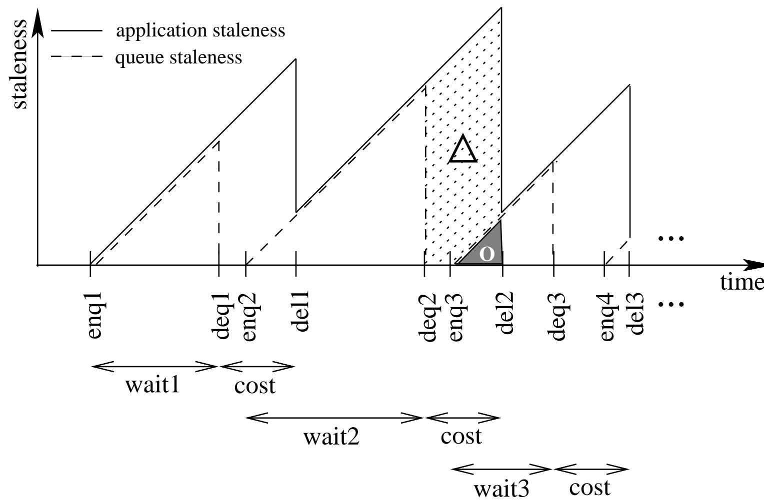
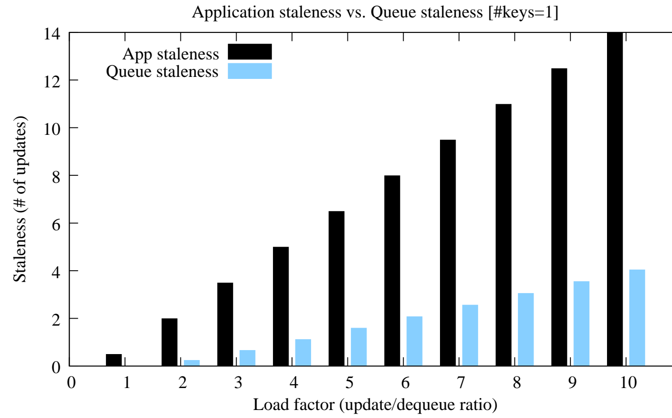


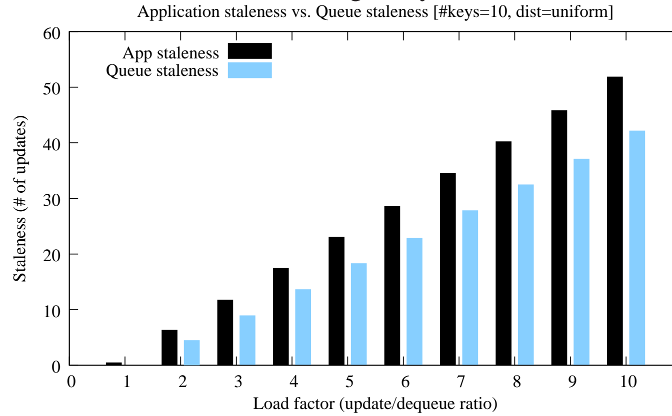
Figure 4: Application staleness vs. Queue staleness

Figure 4 shows two staleness plots for a single update key: (i) solid line for application staleness, (ii) dash 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 get delivered to the application. The overall average staleness would be computed by taking the area under a given plot and then dividing it by the total run time passed. 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 4. 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:

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



(a) Single key



(b) Multiple keys

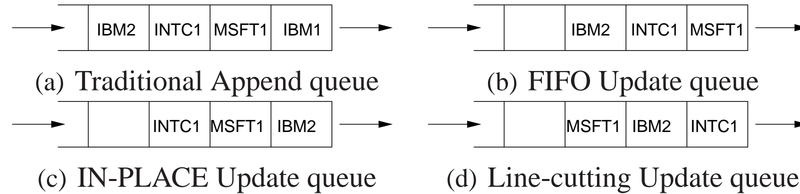
Figure 5: Application staleness vs. Queue Staleness

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 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. Figures 5(a) and 5(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 5(a), in Figure 5(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

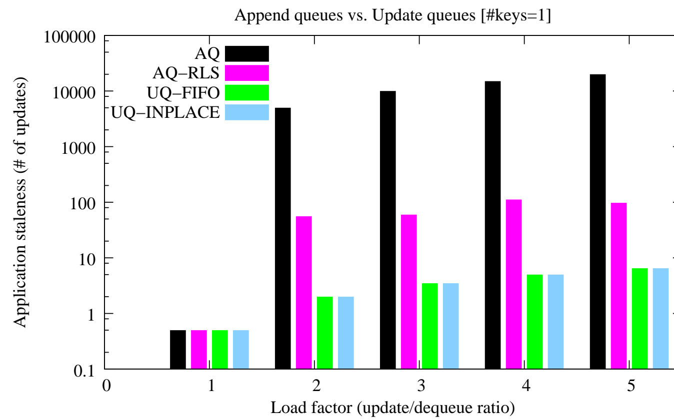
(c) IN-PLACE Update queue (d) Line-cutting Update queue

Figure 6: Update queue examples

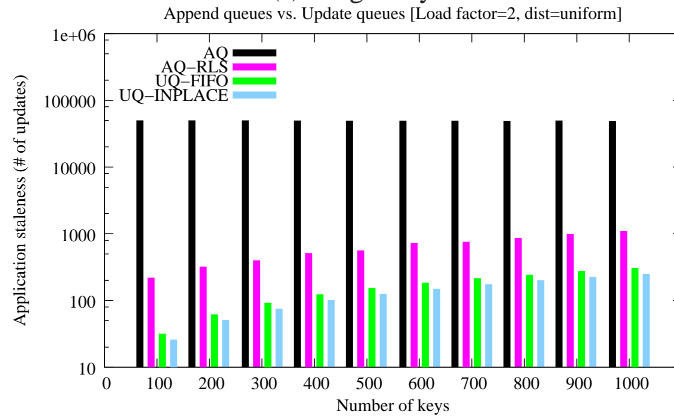
We now introduce two baseline approaches to update key scheduling: FIFO and IN-PLACE.

- **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 6(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 6(c), 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.



(a) Single key



(b) Multiple keys

Figure 7: Append queue vs. Update queue

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). Figure 7 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 $p_1 = p_2 = \ldots = p_n$. 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 8 illustrates this situation for financial market data taken from NYSE Trade & Quote (TAQ) database for a trading day in January 2006 [1]. 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?

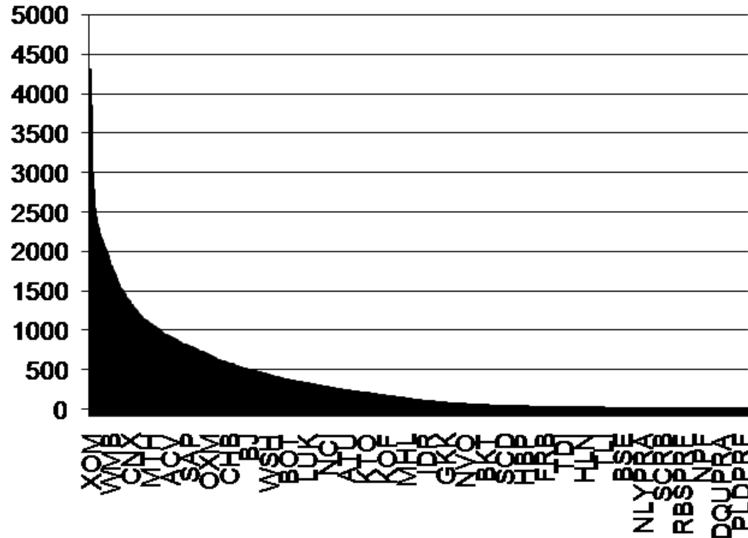


Figure 8: NYSE TAQ data (daily # of updates per stock symbol)

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

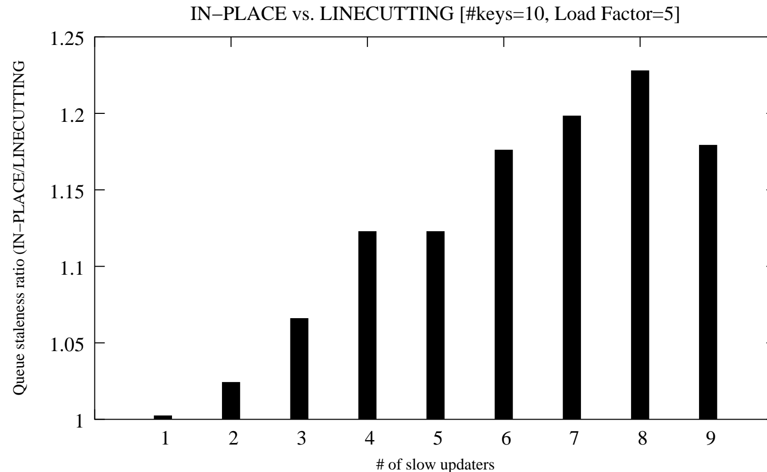


Figure 9: Linecutting

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.

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 9 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 Windowing

In this section, we focus on describing how to apply update semantics for windowing queries. We begin by listing the assumptions and requirements for the window-aware update model. Then, we describe two techniques, EAGER and LAZY, for processing windows under update semantics. Finally, we provide results that show the performance of our windowing techniques.

6.1 Assumptions

6.1.1 Windowing queries

Targeted queries contain operators that define windows over the input stream. We discuss the windowing semantics in the same context as described in [19]. A windowing operator (*Aggregate*) defines windows on the input stream by specifying when to open a window (δ: window slide) and when to close it (ω: window size). The Aggregate operator can apply a series of aggregation functions (e.g: min, max, average, count, etc) on the contents of the window. The computation is applied incrementally as tuples update the window. A state is maintained for each open window that represents the partial computation on what has arrived so far. Windows can be tumbling ($\omega = \delta$) or sliding ($\omega > \delta$) in time. The time taken into consideration by the operator is that of the data source, rather than the system time. Aggregate defines windows on a per-key basis, where the key is the group-by attribute.

6.1.2 QoS model revised

Updates are applied in units of windows. We will show later in the section how we define the most recent update when we introduce the update-based windowing semantics. However, no matter how this is defined, staleness grows with every fully-arrived unprocessed window. The interpretation is that a result can be generated and the application updated only after a full window has been dequeued and processed. The processing cost for a window (that is, for a full update) is expressed differently than in the tuple-based scenario, where the load factor in the system was given by the amount of updates per processing cost per update. In the windowing case, the cost per window (cost for updating the application) C_W must account for the processing cost per tuple C_t, the number of tuples within the window (ω) and the degree of overlapping between windows ($o = [\frac{\omega}{\delta}]$). Ideally, this is given by the following formula:

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

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

In the case of window-aware load shedding, i can vary between 0 and $(o-1)$. Because of this variation, for the windowing update semantics we decided to provide a deterministic cost estimation, which is the lower boundary. This means that the Aggregate will not open a new window until the current one finishes and a result is produced. However, as we will see later on, this does not mean that we do not allow overlapping windows. It is just a matter of postponing the decision until we know for sure that the window we committed to represents the most recent update. Another reason for doing this is in the interest of fairness towards all keys. In conclusion, the cost model for windowing updates has two components: cost for processing a tuple C_t and the window size ω.

6.1.3 Window drops

Our storage-centric approach imposes we shed updates as soon as they become older. Since, in the present case, an update is represented by a window, we need to provide the proper mechanism for storing current window tuples and getting rid of expired ones (belonging to windows that have become old). The update-based windowing semantics must make sure to preserve window integrity when dropping windows. We do this by leveraging the tuple marking scheme proposed by Tatbul et. al [19]. In order to facilitate such a requirement, the update queue and the operator must work together. The update queue keeps track of the most recent window update. To do this, it employs the same mechanism for delimiting windows as used by the operator. When it is time to open a new window for processing, the queue will mark that window with a non-zero window specification. Older windows are marked with 0. On the other side, the operator will open windows selectively based on the window specification marks. In other words, the operator is stripped down of the window delimiting functionality which is better suited in the queue. Table 1 gives the possible actions that the operator might do upon reading a tuple from the queue, based on the win_spec values[4].

6.2 Windowing techniques

We apply the update semantics for windowing consistently with the tuple-based scenario. More specifically, we have to define the most recent update (MRU) rule for windows, which our windowing semantics are based on. For tuple-based updates, the MRU^t rule says that each new arrival (t, x, val_t) for key x overwrites

[4]Throughout this section, we will be discussing windowing semantics applied to one Aggregate operator. Handling more complex queries can be done by taking into consideration the query level window specification as described in [19]

Table 1: Relevant operator actions

t	win_spec	relevant action
	$\tau > 0$	open window $W(start, end)$ where $start = t$ and $end = \tau$, update(W, t)
	0	do not open window, update(W, t)
	-1	update (W, t)
t=end[W]	don't care	emit result for W

the previous unprocessed one ($(t-1, x, val_{t-1})$). For the windowing case, we consider updates in terms of windows. When we process a window for a key, we say that we *commit* to it. It is the job of the windowing-aware update queue to enforce such semantics so as to make sure that the committed windows comply with the MRU rule. Consider that a tuple (t, x) for key x has just been enqueued. The MRU^w rule for windowed updates can be stated in two ways:

1. MRU^w_L: a window W_i is more recent than a window W_{i-1} if both are fully arrived (all the tuples in the window have timestamps less than t) and $Last[W_{i-1}] < Last[W_i] \leq t$. This is the basis for the *LAZY windowing semantics* which will be discussed in detail in section 6.2.1. Basically, under such semantics, the operator commits to a window after it has fully arrived and the update queue drops on window closers.

2. MRU^w_E: a window W_i is more recent than a window W_{i-1} if W_i is partially arrived and $First[W_{i-1}] < First[W_i] \leq t$. This is the basis for the *EAGER windowing semantics* which we will discuss in section 6.2.2. In this case, the operator commits eagerly to the most recently started window even though it hasn't closed so far. The update queue performs drops on window starters.

An important issue that differentiates tuple updates from windowed updates is the fact that for windows a dequeue takes longer (as long as the length of a window). When the operator is ready to serve a key, it will commit to the most recent window for that key, in the sense described by the windowing semantics. Once this is done, the key is extracted from the queue and placed in processing state, at which time the operator starts dequeuing tuples from that window for as long as there are any available (in the EAGER case, a window is committed to even though it hasn't fully arrived). In order to accommodate such a constraint, we extended the KeyScheduler by assigning states to keys. A key travels through the Key Transition Graph (KTG) starting from the moment when the system receives updates for it, to the moment the system updates the application.

We identified the following states to describe the lifecycle of a key:

- ENQ: the start state. In UpStream, the lifecycle of a key starts in the storage with an enqueue.
- WAIT: a key enters this state immediately after enqueue. It is used to buffer keys while other keys are being served. The KeyScheduler maintains a queue of keys in this state - wq. The relative ordering of keys in wq is given by the heuristics employed by the KeyScheduler (e.g IN-PLACE). When a key first enters this state, its queue staleness starts growing.
- READY: once the operator is prepared to serve another key, it extracts the one that lays in the head of the wq list and places it in this state, to mean that the operator is committed to the key. Hence, it is no longer waiting, its queue staleness will go down to 0 and its application staleness will go down once the processing concludes (to a value of 0 only in the absence of pending updates). It may happen that the Aggregate operator has committed to more than one window but belonging to distinct groups (keys). Therefore, we keep a queue of keys that are under processing at the same time called upq.
- IDLE: this state was introduce to strictly accommodate the update-based windowing semantics. The operator will try to consume tuples from a committed window as long as there are any available. Otherwise, the key will be made IDLE in order to make room for other keys that are probably waiting. An idle key will get back in the upq after new arrivals. Idle keys are kept in an unordered list called il.
- DONE: the final state. Once the processing for a key finishes, it is removed from upq. In UpStream, the lifecycle of a key ends with updating the application.

It is important to note that it is possible for a key to be in more than one state at the same time. For instance, in the context of overload, a key may be found in the WAIT and READY/IDLE states at the same time. This means that the key is under processing, but while doing so, new updates have arrived which would eventually cause an increase in staleness.

Figure 10 shows the general Key Transition Graph, independent of the windowing semantics. The conditions for transition between states are as follows: $isUpdate(t, K)$ is a verification of the MRU^w condition. At enqueue, the KeyScheduler uses $doPromote(t, K)$ to check whether K must be reintroduced into READY state. At dequeue, $isAvail(H)$ verifies whether the committed window has more tuples.

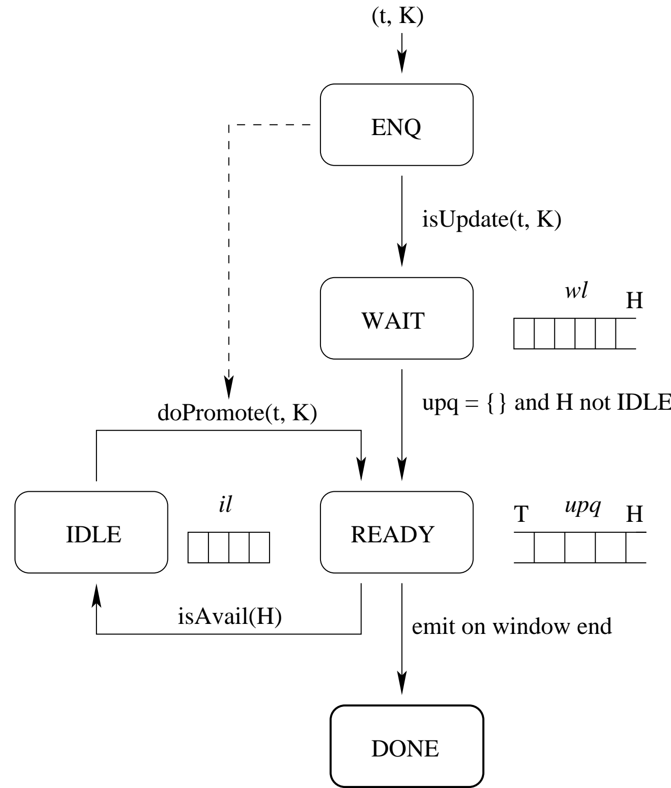


Figure 10: Key Transition Graph

In order to separate between the READY state and the WAIT state at the storage level for one key, we provide an infrastructure for writing, overwriting (updating), deleting and reading during the enqueue and dequeue operations. The fixed part of such an infrastructure - which we call a Window Buffer - is the *dequeue buffer* which holds the tuples of the committed window. The hot part of the WindowBuffer is the *enqueue buffer* where window tuples accumulate as they arrive. Moreover, it is the enqueue buffer that handles drops in terms of windows.

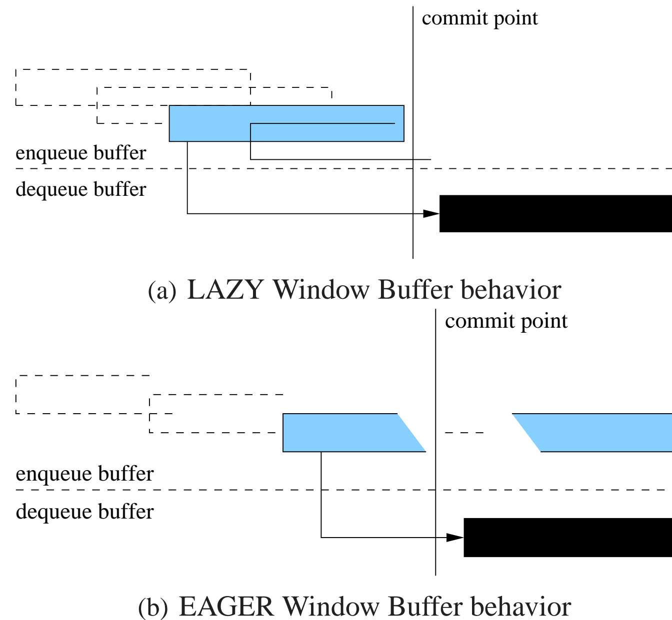
The instantiation of the KTG state transition triggers as well as the Window Buffer flavors will be further discussed in the context of the two windowing semantics.

6.2.1 LAZY windowing semantics

The LAZY windowing semantics are a direct adaptation of the tuple-based update semantics: the operator will only commit to a window that has fully arrived and that is the most recent one, according to the MRU_L^w rule. First, we will show

an instantiation of the KTG state transition triggers for the LAZY windowing semantics. Then, we will give the behavior of the LAZY Window Buffer. Finally we will exemplify with an execution trace.

In the LAZY case, we commit to fully arrived windows. This means that a key in READY state will never run out of tuples and therefore, no key will ever be IDLE. The only thing that we have to define for the LAZY windowing semantics is the instantiation of the $isUpdate(t, K)$: it will return true if and only if t closes a window for key K.



(a) LAZY Window Buffer behavior

(b) EAGER Window Buffer behavior

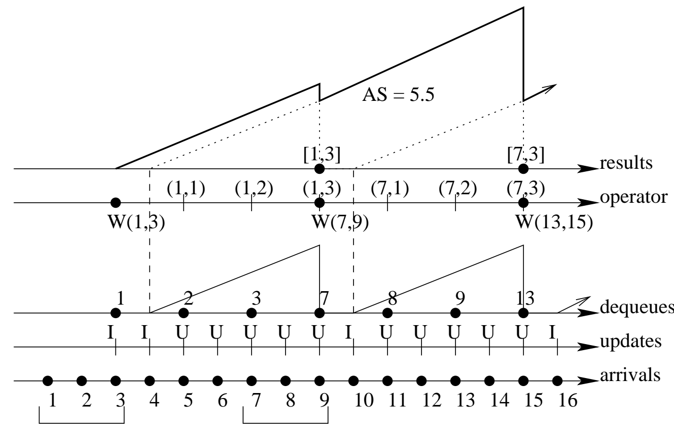
Figure 11: Window buffers

Figure 11(a) shows the behaviour of the LAZY Window Buffer for overlapping sliding windows. The enqueue buffer consists of a list of window structures, one for each window starter that is in the current scope. The window structure that contains the most recent fully arrived window is always at the top of the list. Overwriting happens by replacing the dequeue buffer with the newest top of the enqueue buffer. The event that triggers an overwrite/drop is the occurrence of a window end. The enqueue buffer can contain as many as o window structures [5]. At commit point, the top of the list of window structures in the enqueue buffer is flushed to the dequeue buffer. The tuple corresponding to the window starter in the dequeue buffer is marked according to the schema described in section 6.1.3. All the window starters that are part of the enqueue buffer are marked with 0

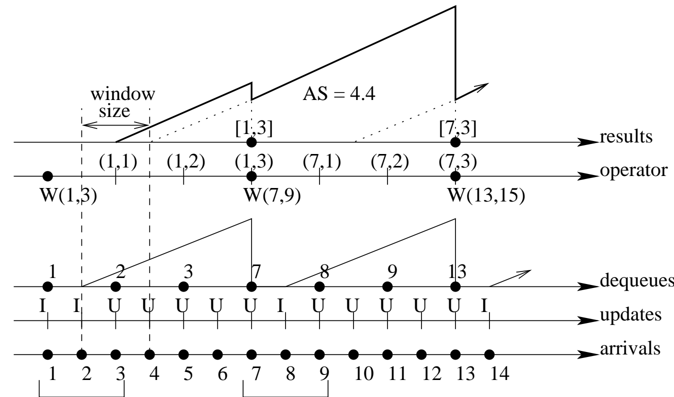
[5] o is the degree of overlapping described in section 6.1.2

(disallow) by default.

Figure 12(a) shows an example of an execution trace for LAZY windowing semantics. We considered one key, $\omega = 3$, $\delta = 1$ and $C_t = 2$. On the dequeue and results timeline, we depicted the queue and application staleness, respectively. I denotes a new arrival for the key, while U indicates an update to the key that is already waiting in the queue. Both queue and application staleness are computed on window ends. Therefore, by the equivalence between tuple-based updates and window-as-a-whole updates, the relationship between queue and application staleness is preserved.



(a) LAZY Window Buffer



(b) EAGER Window Buffer

Figure 12: Execution traces for $\omega = 3$, $\delta = 1$ and $C_t = 2$

6.2.2 EAGER windowing semantics

The EAGER windowing semantics are a combination of append-based windowing and update semantics: the operator commits to the latest started window and eagerly tries to process window tuples as they arrive. As we will show in our example and our experiments, we expect the EAGER approach to outperform the LAZY one in terms of application staleness. However, it is less deterministic than the LAZY approach, since windows may have gaps, especially when there are multiple keys updating at the same time. On the other hand, since updates are considered at each window starter, queue staleness is computed on window starters whereas the application staleness is computed on window closers. We shall prove that the relationship between queue and application staleness is still monotonic [6].

First, we describe the instantiation of the KTG state transition triggers:

- $isUpdate(t, K)$: returns true if and only if t starts a new window for K.
- $doPromote(t, K)$: returns true if K is IDLE, its dequeue buffer contains an incomplete window and t is within the boundaries of that window. This guarantees that a key is back into READY state after hitting a gap.
- $isAvail(H)$: returns true if H has a partially arrived window in the dequeue buffer.

The behavior of the EAGER Window Buffer is shown in figure 11(b). Overwrites happen on window starters. Therefore, the enqueue buffer is always emptied when a new window starter arrives. This means that the enqueue buffer will always contain the most recently started window. At commit point, the enqueue buffer is flushed to the dequeue buffer. The marking is done the same way as in the LAZY case. Subsequent tuples are placed both in the enqueue buffer (as long as they belong to the most recent window that is not committed) and in the dequeue buffer (as long as they are in the scope of the currently committed window).

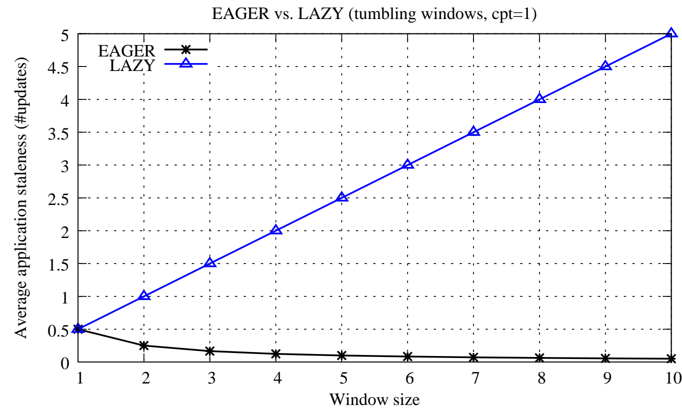
Figure 12(b) shows an example of an execution trace for EAGER windowing semantics. One can observe a mismatch between the absolute values for queue and application staleness. This is because they are computed at different points: window starters and window closers, respectively. However, if we look at the first committed window starting at time 1, the key starts waiting in the queue when a new window starts at time 2. On the other hand, from the application perspective, the window that closes at time 4 invalidates the output for window 1,

[6]In our evaluation of the windowing semantics, we will use only application staleness as a metric for comparison. Queue staleness is a metric to evaluate the heuristics, whereas application staleness offers an evaluation of the whole system.

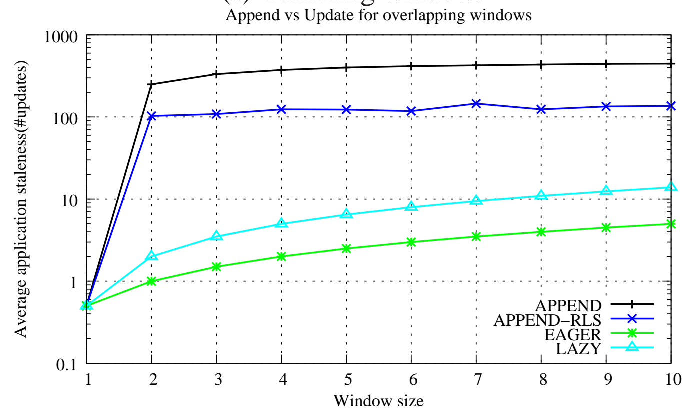
given at time 7. Since the invalidators of both metrics belong to the same window (starter and closer), we can easily conclude that the relationship between queue and application staleness is preserved.

6.3 Evaluation

6.3.1 Windowing semantics



(a) Tumbling windows



(b) Overlapping windows

Figure 13: Windowing semantics

In this section, we will show experimental results on the performance of the update-based windowing semantics. We start off by comparing our strategies with the traditional append style of windowing and a load shedding variation on that. We show this comparison by isolating the windowing semantics effects from those of the heuristics, therefore we are using only one key. Secondly, we will compare

the EAGER and LAZY windowing semantics in the context of multiple updating keys and with different update and arrival patterns.

Here, we want to show the performance of the windowing strategies without worrying about key scheduling and arrival patterns across keys. For these experiments, we considered varying the window size, while looking at tumbling windows ($\delta = \omega$) and overlapping windows which slide at each tick ($\delta = 1$).

In figure 13(a) we compare the effects of the EAGER and LAZY windowing semantics on the application staleness when windows are tumbling. One can see that for the latter one, staleness grows linearly with the window size. This is because windows are processed as a whole only after they arrive, which causes the staleness to account for $C_t * \omega$. On the other hand, for the EAGER case, windows are processed on the fly and staleness grows for C_t time units after the window has closed (in the simple tumbling scenario with $C_t = 1$). This means that staleness will have the same absolute values no matter the window size. Hence, by taking the average while increasing window sizes, staleness for the EAGER case decreases asymptotically.

Figure 13(b) shows a comparison between the append and update windowing semantics. For the append case, we considered the traditional process-everything way and a load shedding variation that randomly drops windows. For the update part, we show results for our windowing approaches: EAGER and LAZY. The results clearly show that the latter outperform even the random window shedding which would drop more or less the same amount of windows as the update semantics would. For this graph, we varied the window size while keeping the slide constant at 1. The random window shedding offers pourer performance with increasing load (window size), since it doesn't try to produce results for the latest windows. Moreover, the randomness in dropping windows may cause the operator to commit to multiple windows at the same time, which falls into the area of the tumbling snowball effect: with each window that the operator opens, the cost for producing a result for previously opened windows grows [7].

6.3.2 Multiple keys and arrival patterns

So far, we have seen that EAGER outperforms LAZY for the 1-key scenario. However, even from the model definition for EAGER, one can anticipate it might perform worse if there are gaps in the windows. A gap in a committed window, means that the key is placed in IDLE state. It will get back in READY state only upon subsequent arrivals within the scope of that window. However, when this happens, other keys may be already in the front of the upq list and this key will be placed at the tail, waiting for the others to finish. In order to examine this

[7] Here, we use no optimizations concerning the window computation.

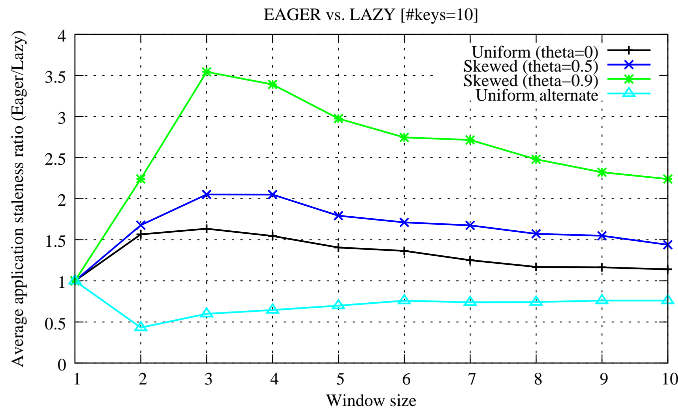


Figure 14: Multiple keys and arrival patterns

possibility, we turned our attention towards various arrival and update patterns across multiple keys.

We considered four distributions: three ZipF distributions generated by varying the θ parameter from 0, 0.5 to 0.9 and a uniform distribution with uniform arrival order among keys (e.g. 1,2,1,2,etc). Figure 14 shows the results of experimenting with these distributions for 10 keys, $C_t = 1$ while varying the window size and keeping the slide fixed at 1. The heuristic we used was IN-PLACE. The comparison between EAGER and LAZY is done by performing a ratio of their application staleness values. The results show that EAGER performs worse than LAZY for the ZipF distributions. This is to confirm our expectations about the problem that the EAGER windowing semantics may have with gaps. One can also observe that the difference between EAGER and LAZY is more prominent in the first part of the x-axis than in the second one. This can be explained by the fact that across runs, the size of the gaps remains the same for the same distribution, but the window size grows which means that the number of updates decreases. Having more updates (for smaller window sizes) causes greater staleness values.

On the other side, one can also observe that for the uniform arrival order distribution, EAGER outperforms LAZY. The reason is that the arrival patterns and gaps are more deterministic and EAGER is in advantage by processing window tuples on-the-fly.

7 Memory Management

Here, we describe the update queue, a storage-centric implementation of update semantics within a stream processing engine. First, we give the overall assump-

tions that underlie the update-based memory management. Then, we go into detail about our mechanisms for performing efficient management of in-place updates for the tuple-based and window-based scenarios.

A query network in our system consists of operators that are connected by arcs. Arcs represent the flow of data between operators. Associated with each arc is a tuple queue. Input tuples from the data sources as well as tuples resulting from operator execution are pushed into these queues. Tuples in a queue are consumed by their destination operators when these operators get scheduled for execution, or are delivered as query results if the tuple has reached to an output arc in the query network.

Generally speaking, any tuple queue within the query network can also be an update queue. However, to make things simple, we present the situation where the update queue applies only on input streams. An update queue is a shared resource between the facility for enqueuing tuples from streams and the Query Processor which consumes tuples from the update queue and feeds them to the query operators.

In order to handle the various aspects of the update-based model, the underlying storage management system implements the following functionalities:

- *key-based stream partitioning*: the input stream has the following schema: (T, K, V), where (K) is the *update key*. T represents the timestamp of the data source that produced this tuple. V is a set of useful values for a key. Virtually, the system views the update stream as a set of substreams for each key.

- *overwriting*: within a substream for a key x, each new tuple $Tuple(t, x, val_t)$ overwrites the previous one ($Tuple(t-1, x, val_{t-1})$).

- *key scheduling*: once in the system, keys must be ordered according to one of the strategies described in the previous section.

- *memory management*: tuples for each key are stored in the main memory. The system Storage Manager preallocates a chunk of main memory (PagePool) from which all tuple queues in the query network are allocated space on a per-page basis. Traditionally, a tuple queue is a FIFO queue which is conveniently stored as a contiguous memory zone split among aligned pages. This enforces a sequential memory access for both reads (dequeues) and writes (enqueues). However, in our update-based model, dequeues do not happen in the arrival order across tuples, as is the case with append queues. This yields a random memory access pattern and to solve this, our Memory Manager implements an on-demand paging mechanism for efficiently keeping track of key data.

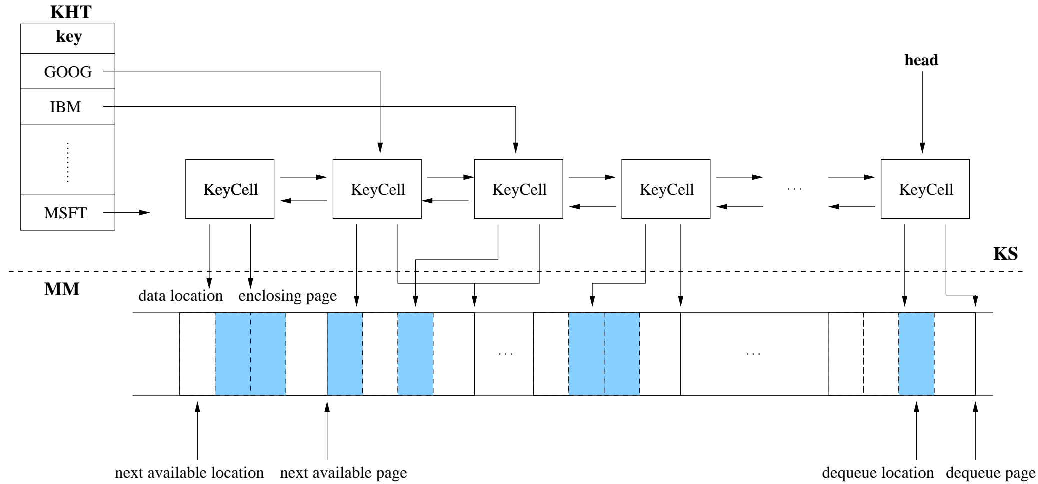


Figure 15: Update queue

The system distinguishes among keys with the help of the KeyHashTable (KHT) (see Figure 15) which assigns a KeyCell data structure to each key. The KeyCell data structure has two roles. First, it stores information about the location in main memory where tuples for a key are stored, so that the access to data is instant. In the case of tuple-based processing, we need only one location for each key. In order to handle this scenario, the KeyCell maintains two pointers: one to the page (*enclosing page - ep*) and one to the location within that page (*data location - dl*). These pointers are used to make in-place updates for new arrivals. On the other hand, the KeyCell allows a decoupling between the underlying storage management layer (MM) and the key scheduling layer (KS).

The KS layer implements various key ordering heuristics, such as FIFO or IN-PLACE. For the purpose of our heuristics, the underlying implementation is a double-linked list of key cells - called Queue of Keys (QoK) - which maintains a pointer to the *head* of the list. The head always points to the key cell that is to be dequeued. next. Adding to this queue is an operation specific to each heuristic. An update queue operates in the following way:

- **Enqueue** (for key “x“). The system asks KHT for the key cell for “x“, adding a new cell to KHT if none is found. Based on the information in the key cell, the system asks the Memory Manager to allocate a new location for this key or use the one already pointed to by the key cell. The former case happens if this arrival for key “x“ is the first after the last dequeue for this key. To facilitate this case, the Memory Manager keeps two pointers: *next available page* (nap) and *next available location* (nal). These pointers are updated after each new allocation by the Memory Manager to indicate

an empty location where we can add tuples. After physically copying the tuple to the computed location, the system calls the Key Scheduler with key "x" in order to add it to the QoK according to a specific heuristic.

- **Dequeue**. The Memory Manager uses the information (page and location) from the head of the QoK in order to access the tuple to be dequeued. After doing so, the Key Scheduler removes the head from QoK.

The Key Scheduler operates according to the heuristics explained in the previous section and the implementation is straightforward. Next, we focus on the memory management mechanism underlying the update queue.

7.1 Paging Mechanism

The problem that arises from implementing a random access memory manager is that it results in page fragmentation. In order to minimize page fragmentation and memory proliferation, we implemented an *on-demand paging* mechanism that allocates only the necessary pages from the PagePool and maximizes space utilization within a page. This mechanism performs garbage collection eagerly, so that when all the locations within a page are freed, the page is returned to the PagePool.

The basis for the on-demand paging mechanism is the use of an additional data structure called the Empty Locations Pool (ELP). This pool stores the pages that are in use at the moment and have available locations. For each such page, the ELP simply stores the list of available locations. Basically, the operation of the on-demand paging mechanism is this:

- *at enqueue* (this only happens if key needs a new location!): grab the first page from ELP and from that page, the first available location stored in ELP. 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. Next, we need to update the next available page pointer to prepare for the next location request.

- *at dequeue*: if the page that we dequeued from was full, it is added to ELP together with the dequeued location. Otherwise, we update the list of available location for this page with the dequeued locations.

This behaviour can be observed in more detail in the pseudo-code below.

```
MemAdd()
 if (nap.isEmpty())
```

```
    if (ELP.isEmpty())
       new_p = PagePool.alloc()
       ELP.add(new_p,{})
       nap = new_p
       nal = first(new_p)
       return
    else
       p = ELP.select()
       nap = p
       nal = pick_avail(p)
 else
    nal= pick_avail(nap)

MemRem()
 if (!ELP.contains(dp))
    ELP.add(dp, {dl})
 else
    ELP[dp].update(dl)
```

The key to ensuring that page fragmentation is minimum is that whenever we pick the next available location from a page (that is in ELP), we always pick one that was already freed. The general policy is to eagerly consume the locations in ELP, which maximizes space utilization within a page.

7.2 Window Manager

Here, we want to introduce the extensions to the storage system in order to support windowing over update streams. Our windowing model assumes the following:

1. Windowing operators perform group-by aggregates on streams.
2. We target sliding windows specified by a size ω and a slide δ.
3. Windows are time-based and the time taken into consideration by the operator is that of the data source, rather than the system time.
4. Windows are defined on a per-key basis.
5. Updates are done in units of windows. The most recent update is the most recently started window. This invalidates the previously started (or closed) window.

6. Staleness grows with every fully-arrived unprocessed window. The interpretation is that a result can be generated only after a full window has been dequeued. Hence, based on the update semantics, staleness grows for every disregarded window.

7. Correctness: we disregard older windows and keep the most recent ones. However, the windowing operators maintain a state by examining the input stream and opening windows at well defined points in time given the size and slide specification. Therefore we need to make sure that we preserve window integrity and we do this by leveraging the tuple marking scheme proposed by Tatbul et. al [19].

8. Efficient memory management: we employ random memory access across windows of different keys while providing a sequential memory access within a window.

Next, we describe the window-aware memory management mechanism that uses the window size and slide specifications to perform window-based updates.

Tuple-based update semantics implies that each individual tuple represents an update to the previous one for the same key. This means that the length of the queue for each key is 1 at all times (except dequeue times, of course, when the queue is empty). The window-based update semantics assumes that we overwrite based on windows rather than on tuples. The queue for one key is now bigger since we must allow window tuples to accumulate before we decide anything about overwriting the current full or partial window. In order to support tuple accumulation for one key, we add a new data structure to the Key Cell which we call a *Window Buffer* (one for each key) and we define an extension to the memory management mechanism in order to efficiently keep track of the Window Buffer contents across memory pages and be able to provide sequential access inside the Window Buffer.

The append-based model assumes sequential access to the memory where tuples are stored: enqueue at the tail and dequeue from the head. If we aligned pages together, we would get a virtually contiguous memory zone between the enqueue location (newest tuple) and dequeue location (oldest tuple). However, in the update-based model, we are dealing with random memory access and, as we have seen in the previous section, we must keep track of the memory locations for each key. In the windowing-based processing, we are faced with the problem of storing multiple tuples belonging to forming windows for one key and keeping track of them in an efficient way. We solve this with the addition of the following artifacts:

1. *Sub-paging layer*: we divide each page into blocks of same size. Each block represents a physically contiguous memory zone where we store tuples with

the property of being consecutive (arrival time-wise) and belonging to the same key.

2. *Block meta-info*: The Window Buffer consists of more such blocks which if we were to align, we would get a virtually contiguous memory zone. A memory block is represented inside a structure which we call a FixedLength-ContiguousZone (in short: FLCZ) that encapsulates the following information about the block:

   (a) enclosing page: the address of the page that contains this block

   (b) block offset: the offset within the page. The address of the block start is computed as $enclosingpage + blockoffset * blocksize$

   (c) dequeue offset: when we extract tuples from a block, we do it from the location corresponding to this offset.

   (d) enqueue offset: when we add tuples to a block, we do it after the location corresponding to this offset. The enqueue and dequeue offsets are incremented with each tuple that is added and respectively, with each tuple that is removed. These offsets enforces the fundamental property of a FLCZ: that of ensuring sequential access inside the block. If the dequeue offset gets past the enqueue offset than the block is empty and it can be deallocated. If the enqueue zone gets passed the maximum number of tuples in a block, then the block is full and we need a new one for future enqueues. Allocation and deallocation of blocks yield a call to the Memory Manager $MemAdd()$ and $MemRem()$. The allocation unit within a page is now a block rather than a single tuple. However, the Memory Manager will apply the same on-demand paging mechanism.

The Window Buffer is a double-linked list of FLCZ structures with an interesting property. By following the NEXT links from a structure to the other in the list and by considering the sequential access property enforced by each structure, then we can define a partial order relation '$<'$' on these structures and the memory locations that they refer. Let (B, L) be a pairing between a block B (represented by a corresponding structure in the WindowBuffer) and a location L within that block. Assuming two pairings (B_1, L_1) and (B_2, L_2), the question is what defines $(B_1, L_1) <' (B_2, L_2)$ as true. If the windowing field is time-like, then it is enough to make the previous statement equivalent to $TS[L_1] < TS[L_2]$, where $TS[L]$ is the timestamp of the tuple stored at location L and '$<$' is the partial order relation defined on ordered natural numbers. This assumption is valid in our model. However, if the windowing field was not time-like, then the '$<'$' relation would have to be defined differently. Maintaining such an ordering is important when

we perform update in terms of windows. The main constraint is to be able to preserve window integrity when we overwrite at enqueue time or block a window for processing at dequeue time.

The Window Buffer is maintained with the help of four pointers. ET points to the block (and location within the block) where we last enqueued a tuple. DF is the block where we dequeue tuples from. WS indicates the most recent window starter and WE indicates where the currently processed window ends or is expected to end. The Window Buffer works like this:

- *at enqueue*: set ET to the current enqueue location. If ET is a window starter, than we erase the currently accumulated window between WS and ET and start a new one. Erasing is not permitted if WS falls inside the scope of a blocked (under processing) window. Regarding the tuple marking mechanism, all window starts are marked with DISALLOW at enqueue. Under these update semantics, the window buffer can reach only a size of maximum $\omega + \delta - 1$.

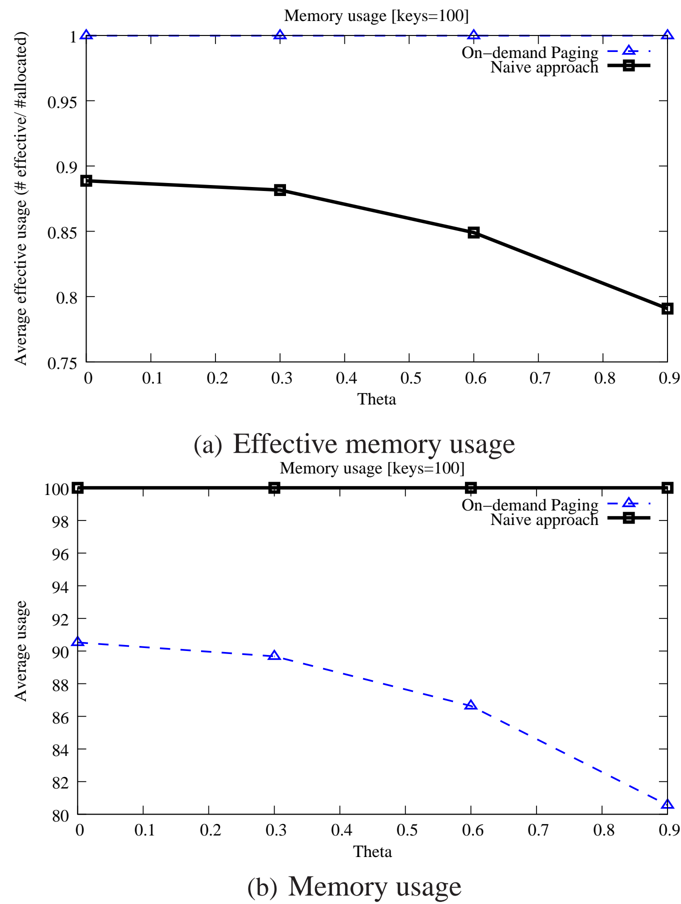
- *at dequeue*: commit to the window indicated by the current WS. This makes it the latest window. At this moment, we give an expected value for WE and we allow tuples to accumulate in the buffer until the window is closed. For tuple marking, the current WS is remarked with ALLOW and the mark is set to the estimated WE.

7.3 Evaluation

In this section, we want to show space efficiency of our memory management techniques through experiments conducted on our prototype system implementation.

In order to have a comparison term, we have developed a naive implementation of memory management for update streams which behaves the following way. When a tuple for a key is dequeued, the Key Scheduler removes it from the queue of keys but the Naive Manager does not release the memory location occupied by the tuple. Therefore, a Key Cell always keeps its memory location. At each enqueue, the Naive Manager must only lookup the Key Cell without trying to compute a new location if that key has no memory allocated. With this approach, the space requirements amount to exactly how many keys we have. This situation is similar to having a preallocated space for all the keys. In the windowing case, the naive approach imposes the existence of a preallocated space for the window buffer. This space would amount to the maximum that the window buffer could reach under the described update semantics.

We compared our on-demand paging approach with the naive approach using the following experimental setup:

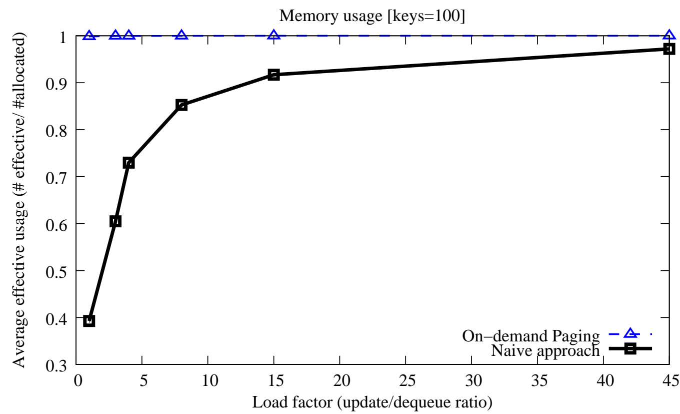


(a) Effective memory usage

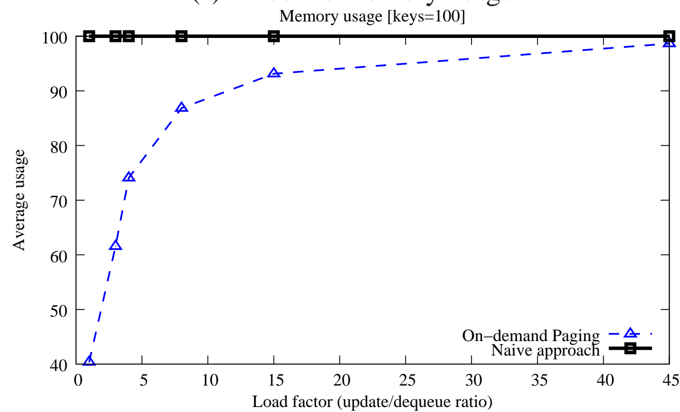
(b) Memory usage

Figure 16: Various update rate distributions

- we used several datasets with uniform and skewed distribution of 100 keys. In order to vary the skewdness between key update rates we used the ZipF distribution. We varied the θ parameter between 0 (uniform distribution), 0.3, 0.6 and 0.9 (very high skewdness).
- 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.
- each experiment runs for the same amount of time equivalent to delivering 1000 results to the application.
- the metrics that we used for comparison are:



(a) Effective memory usage



(b) Memory usage

Figure 17: Various load factors

1. *average usage*: how much space was used on average. The average usage is computed as an average across runs with different distributions.
2. *average effective usage*: the percentage of the total allocated space that was actually in use on average (non-dirty).

We measured average usage and average effective usage while varying the θ parameter (see Figure 16) and the load factor (see Figure 17). 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. When we increase skewdness among the key update rates, this means that the queue is smaller on average than in the uniform distribution case. Figures 16(a) and 17(a) show that the on-demand paging mech-

anism maximizes the usage of the allocated space. A value of 1 means that no matter the distribution, this mechanism does not introduce dirty locations. 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. If we vary the load factor, the effective usage for the naive approach is low for small load factors but it improves while the time between two dequeues increases. If we look at the average usage (Figures 16(b) and 17(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.

8 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 intelligent 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:

- Our experiments focused mostly on minimizing staleness in cases of non-uniform key update frequencies and sliding window queries, and therefore we did not have a chance to generalize our results to more complex queries with arbitrary operator combinations. We need to run further experiments on such queries based on our complete operator set.
- UpStream focused on minimizing staleness for a single continuous query (though 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, similar to the differences in input update frequencies, the application may also want to access the results at different access rates (e.g., would like GOOG stocks to refresh every 1 minutes, while GM stocks to refresh every 1 hour). For this purpose, we need to integrate application access frequencies into our QoS model.
- 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] NYSE Data Solutions. `http://www.nyxdata.com/nysedata/`.

[2] 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.

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

[4] 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.

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

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

[7] 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 (to appear).

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

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

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

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