The VLDB Journal
DOI 10.1007/s00778-011-0229-7

REGULAR PAPER

UpStream: storage-centric load management for streaming applications with update semantics

Alexandru Moga · Irina Botan · Nesime Tatbul

Received: 23 April 2010 / Revised: 17 March 2011 / Accepted: 22 March 2011
© Springer-Verlag 2011

Abstract This paper addresses the problem of minimizing the staleness of query results for streaming applications with update semantics under overload conditions. Staleness is a measure of how out-of-date the results are compared with the latest data arriving on the input. Real-time streaming applications are subject to overload due to unpredictably increasing data rates, while in many of them, we observe that data streams and queries in fact exhibit "update semantics" (i.e., the latest input data are all that really matters when producing a query result). Under such semantics, overload will cause staleness to build up. The key to avoid this is to exploit the update semantics of applications as early as possible in the processing pipeline. In this paper, we propose UpStream, a storage-centric framework for load management over streaming applications with update semantics. We first describe how we model streams and queries that possess the update semantics, providing definitions for correctness and staleness for the query results. Then, we show how staleness can be minimized based on intelligent update key scheduling techniques applied at the queue level, while preserving the correctness of the results, even for complex queries that involve sliding windows. UpStream is based on the simple idea of applying the updates in place, yet with great returns in terms of lowering staleness and memory consumption, as we also experimentally verify on the Borealis system.

A. Moga (✉) · I. Botan · N. Tatbul
Systems Group, Department of Computer Science,
ETH Zurich, Universitatstrasse 6, 8092 Zurich, Switzerland
e-mail: amoga@inf.ethz.ch

I. Botan
e-mail: irina.botan@inf.ethz.ch

N. Tatbul
e-mail: tatbul@inf.ethz.ch

Keywords Data stream processing · Load management · Update streams · Staleness

1 Introduction

Processing high-rate data streams in real time has been a challenge for many applications including financial services, network traffic monitoring, and location tracking systems. Stream Processing Engines (SPEs) have been built to run continuous queries over such streams and to produce results in near real time. A major challenge for these systems is to be able to keep up with data rates and when this is not possible, take the necessary actions to adapt to overloading situations. This paper investigates ways to minimize the staleness of query results for streaming applications with update semantics under conditions of high load.

Although SPEs typically model data streams with append semantics, after careful consideration of several streaming domains such as the ones listed earlier, and not only, we were able to find streams and queries that exhibit **update semantics**. "The latest data are all that matters" can summarize this property succinctly. Furthermore, during our analysis in the streaming domain, we have observed that usually data are aggregated and processed based on certain items of interest. We refer to these as *update keys* and to the nature of queries that preserve update key information as *key-sensitive queries*.

Let us consider an example taken from the financial services domain. We have two data streams, one containing the trades and the other the quotes registered during a trading day in the stock market:

```
Trades(Time, Symbol, Price, Volume)
Quotes(Time, Symbol, Bid, BidVolume, Ask, AskVolume)
```

A trade tuple contains the price and volume of a stock traded for a particular stock symbol at a particular time.

The price is attributed to the *current market price* for stocks belonging to that symbol. A quote tuple contains the highest bid and the lowest ask for a symbol together with their corresponding volumes that are attributed to the *current state* of buy and sell orders for the same symbol on the market at a particular time. These streams clearly have update semantics, and we call them *update streams*.

Streaming queries can also exhibit such update semantics. For instance, consider the following query:

```
Q1: For each symbol, report the price and the change
    in price relative to the beginning of the day.
```

`Q1` is a key-sensitive query whose output stream is an update stream as well, since the results are updates for the same stock symbol, i.e., the key. The window of interest is the time elapsed from the beginning of the day. The following query would be another example:

```
Q2: For each symbol, report the total trade volume and
    average trading price in the last 10 min.
```

Here, the window of interest consists of the last 10 min worth of tuples and slides over the Trades stream.

An SPE running such queries must ensure that query results are delivered in a timely fashion. We can identify two situations:

- The current result reflects the *latest* data: In the context of `Q2`, the average price is computed and delivered before the window gets the chance to slide.
- The current result reflects *obsolete* data: While the current average price was being computed, new trades were registered that caused the window to slide. This happens in situations of **overload** when the system cannot keep up with data rates. In these cases, **staleness** is the metric to measure the amount by which query results are superseded by newer input data. Therefore, when processing streams with update semantics, the main goal is to provide the *most up-to-date answers* to the application with the *lowest possible staleness*, as opposed to streams with append semantics, where providing *all answers* with the *lowest latency* is more important.

The staleness problem has been approached previously in web databases, soft real-time databases, or streaming data warehouses [2,7,10,15,16]. Basically, that research considered scheduling updates to base data or synchronizing local copies of remote databases while analyzing the trade-offs between data timeliness (i.e., staleness) and response time of user queries. We approach the problem of minimizing staleness in the streaming context, which offers a different setting: Updates and queries are part of the same pipeline, data arrives as high-rate push-based streams, while queries are persistent. In stream processing, various load management techniques have been proposed ranging from dynamic load balancing (e.g., [31]) to adaptive load shedding (e.g., [28]). However, none of them have considered the special setting offered by applications with update semantics.

Our key insight into minimizing staleness of streaming applications with update semantics under overload conditions is to push update semantics up the processing pipeline, thereby reacting to load accordingly and as early as possible. We propose *UpStream*, a **storage-centric framework** for load management based on **update queues**. We exploit the fact that update streams naturally lend themselves to load shedding. This allows for smarter load management while preventing queues from growing uncontrollably.

Our basic solution for load management is to perform key-based in-place updates in the queue. We further enhance this solution in two key directions: (i) leveraging stream characteristics (such as non-uniform key update frequencies) for intelligent key scheduling, and (ii) ensuring correct query results and low memory usage while running sliding window queries.

More specifically, this paper presents the following contributions:

- A model for update-based stream processing,
- A storage-centric load management framework based on update queues,
- Two alternative policies for directly minimizing staleness (IN-PLACE and LINECUTTING) based on heuristics that exploit differences in update frequencies across different keys,
- Techniques for correctly and efficiently processing update streams across complex sliding window queries in order to minimize staleness and memory consumption, and
- Experimental performance analysis on a prototype implementation that builds on the Borealis stream processing system [1].

The rest of the paper is organized as follows: In Sect. 2, we describe the basic models and definitions that set the stage for the research problem we address in this paper. Section 3 introduces our storage-centric load management framework. Section 4 describes the IN-PLACE and LINECUTTING policies for minimizing staleness. Section 5 explains how UpStream can process queries with sliding windows in a way to ensure correctness and low staleness, while using the memory resources efficiently. In Sect. 6, we briefly summarize the implementation of our techniques, putting all the pieces together. The prototype-based experimental results are presented in Sect. 7. We summarize the related work in Sect. 8 before we conclude and discuss future directions in Sect. 9.

2 Models for processing update streams

In this section, we first present the general models for update-based stream processing. We build on these models when we later define staleness and contrast it to latency, arguing that staleness is a more suitable metric for applications with update semantics.

2.1 Streams and queries

Let us first define how we model data streams:

Definition 1 (*Data stream*) A data stream is a possibly infinite sequence of data items containing relational tuples of the form (T, K, V), where T is a timestamp for the time of generation at the source, K is the "update key" field, and V holds a payload value.

Our definition can be compared with the stream data models described in previous work [19], where a stream is considered to represent a *signal*, described by a one-dimensional function $A : [a_1 \ldots a_N] \rightarrow R$. The stream is a sequence of data items (a_i, d_i), where a_i is a domain value $(1 \le i \le N)$ and d_i is data associated with the domain value. Each new data item generates a change in the state of the signal $(A[a_i])$. Three models were proposed based on how the change takes place: (i) in the *time series* model, $A[a_i] \leftarrow d_i$; (ii) in the *cash-register* model, data items are increments to the old state, $A[a_i] \leftarrow A[a_i] + d_i$; and (iii) in the *turnstile* model, both increments and decrements are allowed. In our stream definition, update key fields correspond to the domain values. Furthermore, time series change model is the one that best suits the update semantics.

Let us next define how we model continuous queries over streams. Depending on the mode of input and output data delivery, there are four possible continuous query models in general. Figure 1 illustrates these alternatives. We focus on the *push-push* model in this paper. In our model, a continuous query Q applies key-based computation on an input stream I and produces results on the output stream O. Data elements on I and O first need to conform to Definition 1 and then to a relationship determined by the transformation applied by Q.

Definition 2 (*Lineage mapping function*) A key-wise lineage mapping function l maps every output $o \in O$ to a subset of I from which o was derived (denoted by $l(o)$), as a result of the transformation applied by Q. If $o = (t_o, k_o, v_o)$, then $\forall i = (t_i, k_i, v_i) \in l(o) : k_i = k_o$.

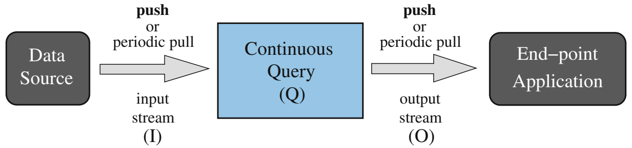


Fig. 1 Continuous query models

The scope of the lineage mapping function [17] or the "lineage set" is determined by the query operations. To the query, the input stream appears as a concatenation of lineage sets. Similar to [11], let us call this the *concatenated stream of lineage sets*, denoted by $\mathscr{P}_l(I)$.[1]

For the rest of the definitions, we introduce two helper functions that are overloaded to cover input and output tuples as well as lineage sets. *key* maps elements to their key value: $\forall o = (t_o, k_o, v_o) \in O : key(o) = key(l(o)) = k_o$. τ maps elements to timestamps: $\forall o = (t_o, k_o, v_o) \in O : \tau(o) = t_o$; for the corresponding lineage set $l(o)$, $\tau(l(o))$ returns the timestamp of the last item in the lineage set.

Here is how we model a stream with update semantics:

Definition 3 (*Update mapping function*) Given two output tuples $o, o' \in O$, o' is the *update* of o over the output stream O if and only if (i) $key(o') = key(o)$ and $\tau(o') > \tau(o)$, and (ii) $\nexists o''$ such that ($key(o'') = key(o)$ and$\tau(o') > \tau(o'') > \tau(o)$). An update mapping function u is defined over the output stream O such that $u(o) = o'$.

The update mapping function allows us to reason about results with the same key value as a stream of time-based updates (rather than value-based ones which would be caused only by changes in the payload value). A new result effectively invalidates the previous one. In fact, the result stream is an *update stream*. Updates in the output stream map directly and uniquely onto elements in the concatenated stream of lineage sets. Thus, $\mathscr{P}_l(I)$ is also an update stream: Given two results $o, o' \in O$ such that $u(o) = o'$, then $l(o')$ is the *update* of $l(o)$ over $\mathscr{P}_l(I)$.

In theory, we can model stream processing using the timeless approach: Imagine having the entire output stream and being able to immediately observe the lineage set of each result. In practice, however, we may need to delimit the lineage set in the input stream without having a corresponding result (e.g., before the query gets the chance to produce it). There are indeed situations when this can be done easily, assuming the query can produce a result based on a finite set of tuples delimited using query parameters. For instance, the case of *tuple-based operations* is simple: Every tuple is an update both in the input and in the output, and the query must preserve the key values. On the other hand, *window-based operations* (e.g., sliding window aggregations) logically group the input stream by group-by fields and construct sliding windows on each substream based on a window size (w) and a window slide (s). One or more aggregate functions are applied on the window, producing an output tuple. The

[1] The notation is inspired by the fact that the lineage sets are among the elements of the power set of I, $\mathscr{P}(I)$.

group-by fields must be a superset of the key fields to be able to retain them in the output.[2] If o_2 is an update on o_1 in the output, then $l(o_2) = w_2$ is an update on $l(o_1) = w_1$. Thus, the unit of update is a window of tuples. The difference in operational units must be taken into account in managing the updates in a real system. This is an important issue that we will discuss again in detail in Sect. 5.

2.2 Order and correctness

In our model, we assume that both input and output update streams are partially ordered, while elements with the same key values are totally ordered with respect to τ values. Despite these ordering assumptions, in a real stream processing system, we can assume nothing about the order in which processing takes place nor the order of result delivery. These would affect the relative order between results and lineage sets, which we consider a matter of correctness in runtime operation. That is, *given two outputs o_1 and o_2 (not necessarily with the same key value), with $\tau(l(o_2)) \geq \tau(l(o_1))$, how are o_1 and o_2 ordered?* We classify system models for stream processing based on this correctness criteria, starting with the definition of an *ideal* system:

Definition 4 (*Instantaneous system* (IS)) An instantaneous system (IS) is an infinitely fast system capable of scheduling any query, processing any stream tuples arriving at any rate, and producing any query results in zero time once necessary data are available.

An IS is a system in which if $\tau(l(o_2))$ is strictly greater than $\tau(l(o_1))$, then $\tau(o_2) > \tau(o_1)$ is always guaranteed. This covers the case when $key(o_1) = key(o_2)$ but not only. That is, it can also happen that $\tau(l(o_1)) = \tau(l(o_2))$, which is only the case when $key(o_1) \neq key(o_2)$. Then, an IS would produce $\tau(o_2) = \tau(o_1)$, which is correct based on the partial ordering of the output stream.

The IS definition offers an idealistic view of the system. In a real system, however, processing a tuple, scheduling a query, or delivering a result take time. A real system may employ various techniques to deal with load or the queries may exhibit different execution semantics that can affect the order in which results are produced.

Definition 5 (*Strictly correct system* (*SCS*)) A strictly correct system (SCS) is one in which a query's outputs having the same key value are the same and appear in the same order as in an IS.

Definition 6 (*Non-strictly correct system* (nSCS)) A non-strictly correct system (nSCS) is one in which a query's outputs having the same key value are a subset of the ones generated in an IS but appear in the same order.

Definition 7 (*Incorrect system* (ICS)) An incorrect system (ICS) is one that does not guarantee the same order of a query's outputs as in an IS, nor does it guarantee the same outputs.

Systems that are not strictly correct deliver approximate results, making them less accurate. For instance, an nSCS can cover subset-based approximation, while an ICS can cover cases such as delivering nearly the same amount of results that may be inaccurate by themselves (this distinction has been made before when debating the correctness of window-aware load shedding in [27]). In this paper, we focus on highly loaded non-strictly correct systems (nSCS).

2.3 Staleness

In a SCS, all input tuples are expected to be processed completely (append semantics). 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. Therefore, end-to-end processing latency is the most important QoS metric.

In a nSCS, the completeness requirement for the result set is relaxed and a different set of challenges arises. That is: *How does a nSCS explain the results and/or lack thereof to the Application?* One way to report on QoS degradation is to use latency and accuracy loss (e.g., number of results delivered per number of results that would have been produced by a SCS) [5,25,27–30]. In this paper, we study applications with update semantics for which the significance of a result is directly stated: *It is more important to deliver the most up-to-date result than all results with low-latency*. To capture this goal, we introduce a different QoS metric, *staleness*.

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 [4,10,21,15]. 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 this work, we use a *time-based definition* for staleness, since this captures the real-time aspect of our target applications more directly. Furthermore, a time-based staleness metric facilitates contrasting or integrating update semantics with the traditional append semantics that is typically based on latency, which is also a time-based metric.

Staleness is a property we associate with each output substream that we deliver for each different update key value through a continuous query. It shows how much a result tuple

[2] In the rest of the paper, for ease of presentation, we simply use queries where the group-by fields are the same as the update key fields; our techniques are general enough to handle the superset case.

with a certain update key value k falls behind in time with respect to more recent input arrivals for k.

Next, we need to adapt the definitions of lineage and updates for subset-based query approximation in a nSCS:

- In a nSCS, only a correct subset Θ of all the results in O gets produced.
- For every output update $o \in \Theta$, there exists a set of tuples in I, denoted by $\lambda(o)$, based on which o was produced by Q and $\lambda(o) = l(o)$.
- Given two results $o, o' \in \Theta$, o' is the update of o, that is, $u(o) = o'$ over Θ, if there exists a set of m tuples, $o_{di} \in O, (i = 1 \ldots m, m \geq 0)$, such that $u(o) = o_{d(1)} \wedge \cdots \wedge u(o_{d(i)}) = o_{d(i+1)} \wedge \cdots \wedge u(o_{d(m)}) = o'$ hold over O.

Now, we can introduce the formal staleness definitions:

Definition 8 (*Staleness of a result*) For any output update $o \in \Theta$, we define its staleness as:

$$S(o) = \begin{cases} \tau(o) - \tau(l(o_d)), & \text{if } \exists o_d \in O\, s.t.\, u(o) = o_d \text{ and} \\ & \tau(l(o_d)) < \tau(o) \text{ hold over } O \\ 0, & \text{otherwise.} \end{cases}$$

The staleness of a result tuple indicates the amount of time by which the result has been superseded by more recent (fresher) input updates, at the instant in time when it was produced. We can view the staleness of a result as the penalty the system has to pay for committing to processing an input update while fresher updates await (either before the start of processing or after). By contrast, latency of the result is computed as $(\tau(o) - \tau(l(o)))$, that is, the time elapsed since necessary data arrived in the input.

Definition 9 (*Staleness of a key*) The staleness of an output update stream for a key k is a function of time, $\mathscr{S}_k$, characterized on time intervals between any $o, o' \in \Theta$ with $u(o) = o'$. The staleness of the key at any time t in the interval $[\tau(o), \tau(o'))$ is given by the following formula:

$$\mathscr{S}_k(t) = \begin{cases} t - \tau(o) + S(o), & \text{if } S(o) \neq 0 \\ 0, & \text{if } S(o) = 0 \text{ and} \\ & t \in [\tau(o), \tau(\lambda(o'))) \\ t - \tau(\lambda(o')), & \text{if } S(o) = 0 \text{ and} \\ & t \in [\tau(\lambda(o')), \tau(o')). \end{cases}$$

The staleness of a key evolves in time as follows:

- If the result is stale when it is produced ($S(o) \neq 0$), it will become even more stale as time passes (as it ages in the output) until new results are produced (o') with lower staleness values. Staleness of a key at time t can be interpreted as the staleness of the last result tuple ($S(o)$) plus the age of the result, i.e., how much time it has spent in the output without being updated ($t - \tau(o)$).
- If the result is not stale, i.e., it reflects the freshest input data ($S(o) = 0$), it will stay fresh (zero staleness) until new input updates arrive ($\lambda(o')$), at which time the staleness of the key starts to grow again.

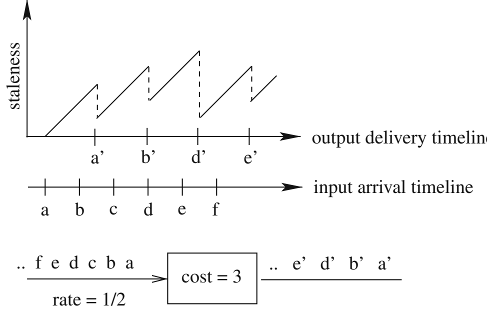


Fig. 2 Staleness

Figure 2 illustrates our staleness definition. In this example, there is one continuous query (simply with tuple-based operators) and a single update key value observed in the input stream. Update tuples (e.g., a) arrive at a rate of 1 update per 2 time units, while results are produced every 3 time units (e.g., a'). Note that $a = \lambda(a'), b = \lambda(b')$, and so on. As soon as a new input update arrives, staleness (of the key) starts to increase linearly with time. For example, update b arrived while update a was being processed. When a' is delivered, staleness goes down to ($\tau(a')$ - $\tau(b)$). The query starts processing b, but staleness increases with time. When b' is finally delivered, staleness goes down to ($\tau(b')$ - $\tau(c)$). At this point, both c and d have arrived. To lower staleness, the system should skip c and pick d for processing, which is more recent. When d' is delivered, staleness substantially goes down, but does not become completely zero, since e has arrived a short while ago. Here, we can see the difference between staleness and latency. Latency captures how long it takes to produce an output (e.g., for d', the latency is $\tau(d') - \tau(d)$), whereas staleness captures how out-of-date an output (e.g., d') is because a newer tuple (e.g., e) has superseded it in the input (e.g., for d', the staleness is $(\tau(d') - \tau(e))$).

Definition 10 (*Staleness of a stream*) Staleness of an output update stream with n keys is a function of time, $\mathscr{S}$, characterized as follows: $\mathscr{S}(t) = \mathbb{F}(\mathscr{S}_{k_1}(t), \ldots, \mathscr{S}_{k_n}(t))$, where $\mathscr{S}_{k_i}(t)$ is the staleness of the key k_i $(i = 1..n)$ and $\mathbb{F}$ denotes a generic user defined function of n parameters.

This definition offers a flexible framework for measuring staleness of an entire stream at each moment in time. However, staleness of the result tuple and staleness of the key remain the invariants of our framework, as they express the properties of applications with update semantics. We left the door open for the application to define a specific way of

aggregating staleness values at the stream level. In previous work, we have seen several ways of doing so. Reporting on the maximum staleness value, averaging over staleness values of all keys or showing variability are among them. In this work, we focus on aggregating staleness values using an average function so that we can reason about the efficiency of our techniques in a more concrete way. The *average staleness of a key* $\overline{\mathscr{S}}_k$ and the *average staleness of the entire stream* $\overline{\mathscr{S}}$ of n updating keys for the interval $[t_1, t_2]$ can be computed as follows:

$$\overline{\mathscr{S}}_k([t_1, t_2]) = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \mathscr{S}_k(t) \mathrm{d}t,$$

$$\overline{\mathscr{S}}([t_1, t_2]) = \frac{1}{n} \sum_{i=1}^{n} \overline{\mathscr{S}}_{k_i}([t_1, t_2]).$$

3 Storage-centric approach

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. 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 easy and efficient to capture update semantics as part of a tuple queue. Applying "in place updates" in the queue is straightforward 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" and others as "update queues" without making any changes in the query processing engine. This kind of model is also in agreement with recently proposed streaming architectures that decouple storage from query processing, such as the SMS framework [8].

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

Traditional append-based query processing models tuple queues as append-only data structures with FIFO semantics. To support update semantics, 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 only keeps the most recent unit of update worth of tuples; older

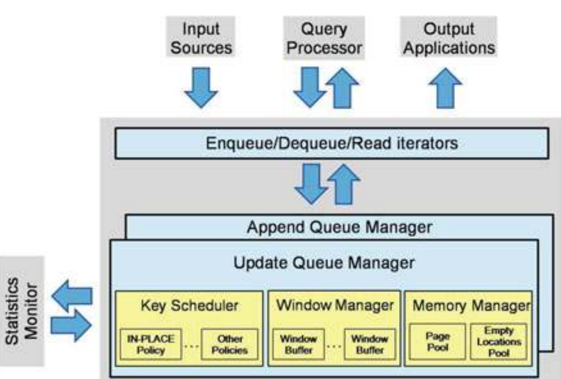


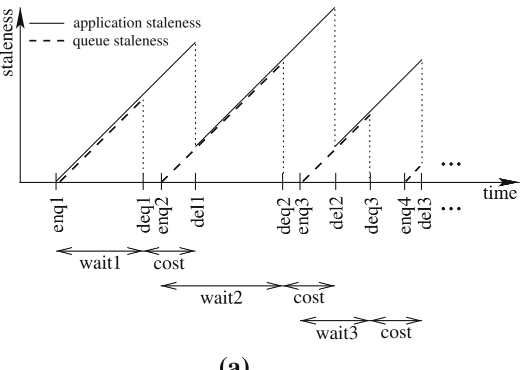
Fig. 3 UpStream storage manager architecture

ones are discarded.[3] An update queue has one data cell per update key value, which holds tuples for that key value, i.e., the most recent update. Nevertheless, 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.

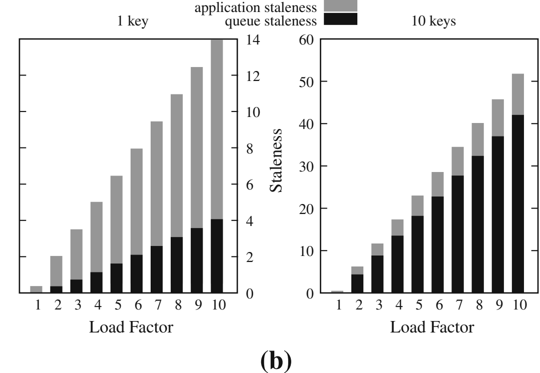
Figure 3 shows the architecture of our storage manager. The UpStream Storage Manager interfaces with the Input Sources, the Output Applications, and the Query Processor through its iterators. These iterators enable three basic queue operations: *enqueue*, *dequeue*, and *read*. Input Sources enqueue new tuples into a queue, whereas Output Applications dequeue tuples from a queue. The Query Processor can enqueue intermediate results from operators, while it can read or dequeue these back again to feed into the succeeding operators in the query pipeline. The Storage Manager also communicates with the Statistics Monitor in order to get statistics to drive its optimization decisions. The underlying queue semantics can be either append or update. In this paper, we focus on the latter.

The Update Queue Manager is divided into three main components. The Key Scheduler (KS) decides when to schedule different update keys for processing and can employ various different policies for this purpose. 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

[3] For ease of presentation, we will first focus on tuple-based queries where the unit of update is a single tuple; extensions to window-based queries will be provided later in Sect. 5.



(a)



(b)

Fig. 4 Application staleness versus queue staleness. **a** Staleness for a single key. **b** Experimental verification on a simulator

of the available system memory where the actual data are physically stored.

4 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. We first introduce a system-level staleness metric that can be more directly and easily used in our algorithms. Then, we introduce our base key scheduling policy, IN-PLACE, and show that it is the best policy for the case of uniform update key frequencies. For the non-uniform case, we propose the LINECUTTING policy. Finally, we experimentally compare the two policies to reveal the staleness improvement that can be achieved by the LINECUTTING policy under various load scenarios.

4.1 Application staleness versus queue staleness

As shown in Sect. 2.3, 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. Next, we show that one can control application staleness by controlling queue staleness and vice-versa.

Figure 4a shows two staleness behaviors for a single update key: (i) solid line for application staleness and (ii) dashed line for queue staleness. The x-axis represents the run time where we depict the arrival times of updates for a single key ($enq1, 2, 3, 4$), the dequeue times to the Query Processor ($deq1, 2, 3$), and the result delivery times to the application ($del1, 2, 3$). Let us look at what happens between $del1$ and $del2$. Based on Definition 9, staleness of the key at any t between these points is $(t - del1 + S(del1))$. $S(del1)$ matches the waiting time of newer updates than that from $enq1(S(del1) = wait1 = del1 - enq2)$. In order to reduce staleness between $del1$ and $del2$, our system should deliver the result for the key sooner, i.e., smaller $del2$. In our model, we assume the cost of processing an update is fixed, i.e., $del1 - deq1 = del2 - deq2 = del3 - deq3 = cost$. Therefore, reducing (application) staleness in our example means to dequeue sooner, i.e., smaller $deq2$. This effectively reduces the waiting time of a key in the queue ($wait2 = deq2 - enq2$) and hence its queue staleness.

We have also experimentally verified our above analysis on a simulator. Figure 4b 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 (equally spaced in time) scales with increasing load. In our simulator, load is modeled as the ratio between the number of arrivals and number of dequeues and equals to the cost of processing an update. This is captured by the *load factor* that in the simulator is a controlled variable. In both cases, staleness grows in a similar way, and the difference between the two staleness metrics is determined by an amount, which is solely a linear function of the load factor. Please note that compared with the single key scenario, multiple keys cause the queue staleness to be inflated by about a factor of 10, since there are 10 distinct update keys that have to wait for each other in the queue. This increases the queue

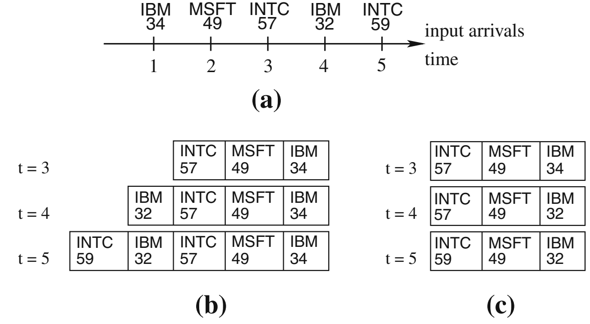


Fig. 5 Append queue versus IN-PLACE update queue. **a** Stock update stream. **b** Traditional append queue. **c** IN-PLACE update queue

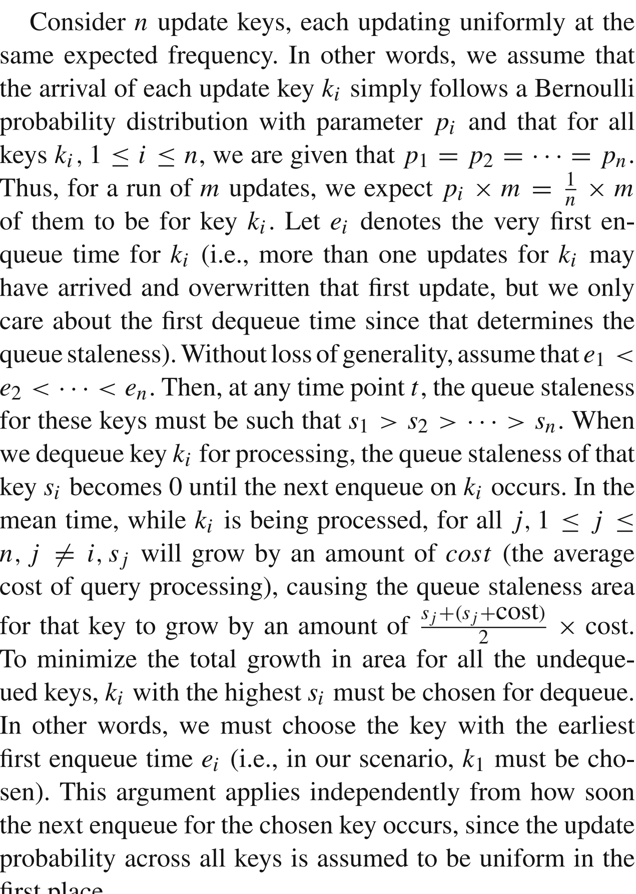
waiting time by a factor of 10. However, the number of keys does not affect the difference between application and queue staleness, which agrees with our theoretical analysis.

This result motivates us into investigating various strategies for scheduling keys to be processed with the goal of directly reducing queue staleness.

4.2 IN-PLACE update queues

We now introduce the "IN-PLACE update queue", our base key scheduling policy. An IN-PLACE update queue is one that stores only the most recent updates for each key group and services them in FIFO enqueue order. In Fig. 5, we illustrate the behavior of the IN-PLACE update queue (Fig. 5c) side by side with the traditional append queue (Fig. 5b), given a stream of stock price updates (Fig. 5a).

When updates first arrive for a certain key, the update queue allocates a *place-holder* to hold any subsequent updates. The place-holder contains both a position for that key in the list of currently updating keys and the necessary memory locations to hold the latest updates (tuple- or window-based). For instance, at time $t = 4$, the price 32 overwrites the previous 34 for an IBM stock in its original place in the update queue, while the append queue would have required a new location at the end. The same thing happens at time $t = 5$, when a new update for key INTC (59) arrives and replaces the previous price (57). If processing occurs right after time $t = 5$, the update queue has the most recent values for all update keys. The next value to be processed would be IBM with price 32, which is the latest arrival for update key IBM. The next value to be processed in the append case would be IBM with price 34, regardless of the newer value 32 being already in the queue. We can see from this example that a given key group in an IN-PLACE update queue does not waste the time it has already spent waiting in the queue if it gets superseded by a newer value. This way, update queues can reduce staleness for key groups. Next, we analyze the performance of IN-PLACE key scheduling.



Consider n update keys, each updating uniformly at the same expected frequency. In other words, we assume that the 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 $p_1 = p_2 = \cdots = p_n$. Thus, for a run of m updates, we expect $p_i \times m = \frac{1}{n} \times m$ of them to be for key k_i. Let e_i denotes 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 $e_1 < e_2 < \cdots < e_n$. Then, at any time point t, the queue staleness for these keys must be such that $s_1 > s_2 > \cdots > s_n$. When we dequeue key k_i for processing, 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* (the average cost of query processing), causing the queue staleness area for that key to grow by an amount of $\frac{s_j + (s_j + \text{cost})}{2} \times \text{cost}$. To minimize the total growth in area 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 independently 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 earlier. 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 waiting time among all keys. Thus, the IN-PLACE key scheduling policy minimizes the maximum waiting time W across all key groups. As a result, IN-PLACE key scheduling ensures that the overall average staleness (application as well as queue) is minimized for a uniform distribution of update key frequencies.

4.3 LINECUTTING update queues

The previous section showed that IN-PLACE key scheduling policy is the best if all keys update at the same frequency. However, often times the update frequencies are not uniform. Figure 6a illustrates such a situation for financial market data taken from the NYSE Trade & Quote (TAQ) database for a trading day in January 2006 [20]. More than 3000 different stock symbols (i.e., update keys) were involved in trading. We can see the update rates being very skewed. Some symbols recorded intense activity throughout the day, while others updated only once or twice in the same day.

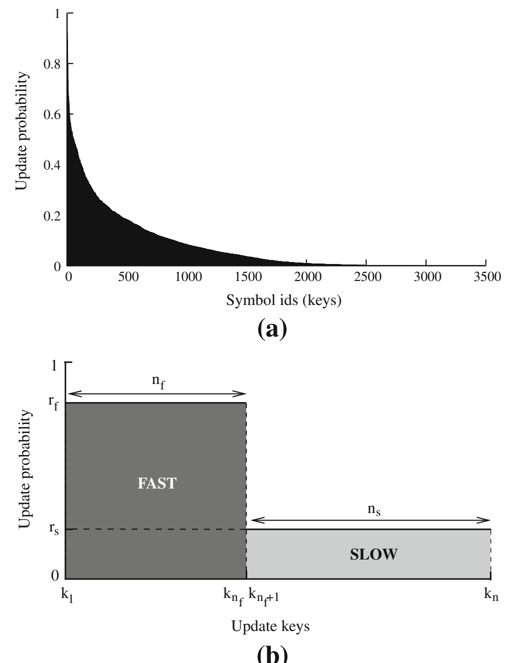


Fig. 6 Non-uniform update frequencies. **a** NYSE TAQ data. **b** Probability distribution with two classes of keys

Based on these considerations, we raise the following questions:

1. Is IN-PLACE policy still the best solution for such non-uniform update frequencies?
2. If not, how can we exploit the differences in update frequencies to find a better key scheduling policy?

4.3.1 Motivating example

Before we address the questions raised in the previous section, we first present an example of a simplified update frequency distribution containing only two keys: Key 1 is fast updating, while key 2 is slow updating. The goal of this example is to explore whether, for such distributions, it is sometimes better to serve keys in an order different from the IN-PLACE order. To this extent, we consider an alternative policy called PROMOTE-SLOW-UPDATER, which is biased toward slow updaters. That is, in the scenario of our simplified distribution, when key 2 enters the queue (the first enqueue after a dequeue), the policy will promote it to the front of the queue so that at the next scheduling point, key 2 will be processed.

Figure 7 shows a comparison between the behavior of the IN-PLACE policy (Fig. 7a) and that of PROMOTE-SLOW-UPDATER (Fig. 7b) in the following setting. The cost of processing a key is $C = 4$ time units. The key update rate is represented as the number of updates per C time units. Consequently, key 1 updates at a rate $r_1 = 2$, i.e., two updates between adjacent scheduling points. Contrary to key 1, key 2 updates much more rarely, at $r_2 = 1/4$. The dotted rectangles mark the periods, called *contention regions*, when both keys are competing to be served. Let us focus on the first of these regions. While key 1 has already been updating at a fast pace, key 2 first arrives between scheduling points 3 and 4. IN-PLACE positions key 2 at the back of the queue, where the key waits to be served at scheduling point 5. PROMOTE-SLOW-UPDATER, instead, promotes key 2 to the front of the queue. This way, key 2 is scheduled at point 4, postponing key 1 for scheduling point 5. In Fig. 7, we have also depicted the evolution of the queue staleness for both keys. We can see that for the contention regions, PROMOTE-SLOW-UPDATER achieves about the same staleness area for key 1, while for key 2, it manages to reduce staleness drastically.

Let us consider a key scheduling point T_s (such as 4 or 8 in Fig. 7b) when both keys are in the queue and let $e_2(T_s)$ be the next update/enqueue time for key 2 with respect to T_s. Then, in order for PROMOTE-SLOW-UPDATER to work as it is, one needs to have:

$$e_2(T_s) - T_s \geq C \times pos(k_2, Q),$$

where $pos(k_2, Q)$ indicates the zero-offset position of key 2 in the update queue Q, i.e., the number of keys in front of

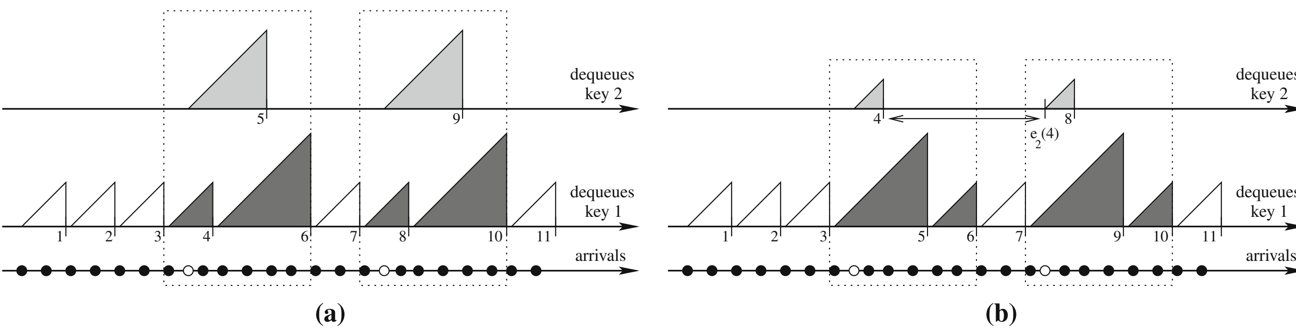


Fig. 7 IN-PLACE versus PROMOTE-SLOW-UPDATER for a trace of updates for two keys (*filled circles* key 1 (fast), *empty circles* key 2 (slow)) **a** No promotion (IN-PLACE). **b** With promotion (PROMOTE-SLOW-UPDATER)

it (in this case only one). From this simple example, we can draw the conclusion that, unless the contention regions overlap, one can indeed benefit from promoting slow updaters. Although simple in its setting, this example motivated us to look further into a policy beyond our base IN-PLACE policy. However, blindly promoting slow updaters may not always pay off. We discuss this in more detail in the next section.

4.3.2 LINECUTTING heuristic

In order to generalize from the aforementioned example, let us consider a probability distribution for update key frequencies as the one in Fig. 6b. We have an update stream with n updating keys split between two classes: n_f are *fast updaters* while n_s are *slow updaters*. Just like in Sect. 4.2, we assume that the arrival of each key follows a Bernoulli distribution with parameter p_i, $1 \leq i \leq n$. In this case, we have $p_j = r_f$ for $1 \leq j \leq n_f$ and $p_k = r_s$ for $n_f + 1 \leq k \leq n$. That is, all fast keys update with the same probability r_f and all slow keys with the same probability r_s. We also assume that $r_f = \sigma \times r_s$ where the *skew* parameter σ indicates how much more a fast key updates compared to a slow key. Given that $n_s \times r_s + n_f \times r_f = 1$, we can obtain the update probability of a slow key: $r_s = 1/(n_s + \sigma \times (n - n_s))$. Based on this, our probability distribution can be fully captured with three parameters: (n, n_s, σ).

To improve on our base key scheduling policy (i.e., IN-PLACE) for the two-class probability distribution, we introduce a key scheduling heuristic called LINECUTTING. The heuristic is based on what we learned from the PROMOTE-SLOW-UPDATER example of the previous section. More specifically, we have designed our heuristic with the following requirements in mind:

1. It should be able to identify the slow updaters with respect to the current state of the queue.
2. Promoting keys to the front of the queue should not lead to starvation of the keys already in front.

Based on these requirements, the goal of the LINECUTTING heuristic is to minimize the maximum quantity $(s+W)$ across all key groups. Here, s is the "slowness" of a key computed based on its update rate r and related to its current position in the queue, and W is the time the key has waited in the queue. Slowness of a key k can be defined more precisely as follows:

$$s(k) = \begin{cases} \frac{1}{r} - C \times pos(k, Q), & \text{if } \frac{1}{r} > C \times pos(k, Q) \\ 0, & \text{otherwise.} \end{cases}$$

The slowness part of the sum $(s + W)$ was suggested by the results of the PROMOTE-SLOW-UPDATER policy, and it accounts for the first design requirement explained earlier. That is, the LINECUTTING policy actively promotes keys that ensure the greatest estimated time until they arrive again in the queue ("the slowest of all the slow ones"). However, as a safety measure against starvation of other keys (faster ones), we also considered W to account for the second design requirement. We would like to make a few notes here: (i) It is highly unlikely to ever promote a fast updater in steady state, while taking into account both slowness and waiting time. For such keys, slowness is approximately zero, which makes the $s + W$ sum reduce to W. This is exactly the same behavior as in the IN-PLACE case. (ii) If the distribution is not skewed, the heuristic acts just like IN-PLACE (slowness would be the same for all keys). (iii) Even though promoted keys may be taken from the middle or the end of the queue, the update queue would still be ordered by first enqueue time.

Let us make a note here about the scheduling process. IN-PLACE key scheduling policy incurs a processing time proportional to $O(1)$ in terms of total number of keys that exist in the queue ($n_q \leq n$) for both enqueue and dequeue operations. LINECUTTING achieves the same time for an enqueue, but $O(n_q)$ for dequeue due to the fact that a queue traversal is needed to find the key with the maximum $(s+W)$ value.

4.3.3 IN-PLACE versus LINECUTTING

Next, we experimentally compared the two key scheduling policies using the simulator. We wanted to observe the clear benefits of LINECUTTING over IN-PLACE without worrying the challenges offered by a real stream processing system. These issues will be addressed later in Sect. 7.2.2. We varied the following factors in our experiments:

- *Number of slow updaters* (n_s) and the *skew* parameter (σ) that characterize our probability distribution for n keys.
- The load in the system (*load factor* LF) caused by a higher input rate than the system can process. We obtain different levels of LF by varying the cost C of processing an update (represented by a single tuple).

We ran several experiments in the simulator that placed LINECUTTING and IN-PLACE side by side. Figure 8 shows results of the comparison based on two representative experiments. In both experiments, we varied the parameters of the distribution while observing the improvement (or lack thereof) of the LINECUTTING heuristic over IN-PLACE for three load levels: low (5), medium (15), and high (25). The y-axis shows the ratio between the average queue staleness achieved by our two heuristics. The horizontal line at $y = 1$ in each graph shows the threshold above which the improvement in staleness starts to be observed.

The first experiment (Fig. 8a) was conducted on a symmetric distribution of 20 keys, out of which 10 keys were

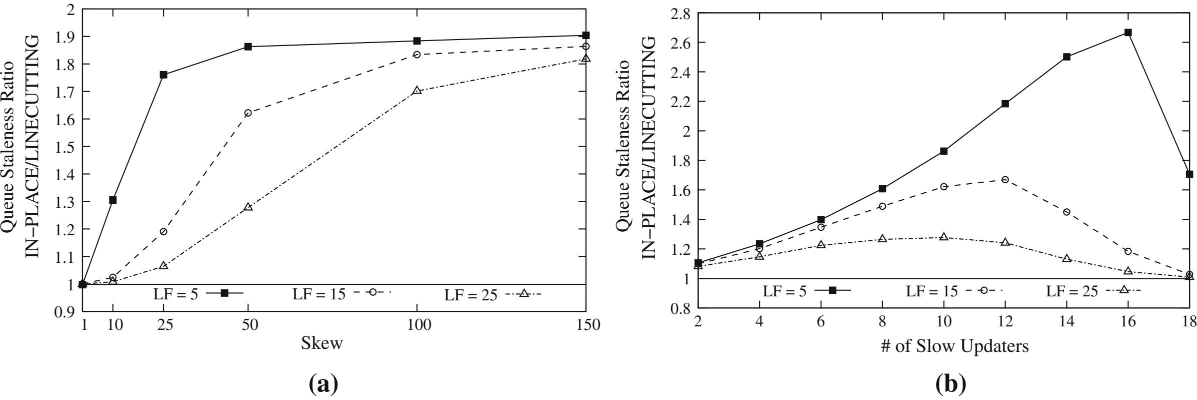


Fig. 8 IN-PLACE versus LINECUTTING. **a** Improvement for symmetric distribution (10 slow, 10 fast). **b** Improvement for asymmetric distributions at skew = 50

updating fast and 10 slowly. We varied the distribution of the keys by varying the skew parameter σ (the x-axis) while n_s was kept fixed at 10. We used this symmetrical case to isolate the uneven contributions of the two classes of keys in the overall average staleness value. The experiment revealed the following:

- Under constant load (e.g., LF = 5), as the distribution became more skewed, the number of updates for slow keys decreased. In terms of the slowness formula, this means that $\frac{1}{r}$ for slow updaters increased, causing them to be promoted sooner after entering the queue and with less penalties on the fast updaters. As LINECUTTING benefits mainly from slow keys being promoted, we can draw the conclusion that the heuristic performs better and better with skew.
- When the load was increased (LF = 15,25) at constant skew (e.g., 50), the $\frac{1}{r}$ part remained the same for the slow keys while the second term of the slowness formula increased, due to $C = LF$. This led to slowness contributing less to slow updaters being promoted, causing them to wait more. Apparently, increasing the load limits the benefits of the LINECUTTING heuristic.

Another point to note is the lack of improvement at $\sigma = 1$ (no skew). That is, LINECUTTING makes no unnecessary promotions, reverting to the baseline behavior, i.e., IN-PLACE, when keys are uniformly distributed. This is primarily due to taking the waiting time into consideration as well when making decisions.

The second experiment (Fig. 8b) was set out to observe the improvement that LINECUTTING may introduce when the number of slow updaters (n_s) is varied at a constant skew (50 in this case). We considered asymmetric distributions of 20 keys with 2,4,6,…,18 slow updaters. We can see that increasing the load had the same effect as in the symmetric case. By contrast, at constant load (e.g., 15), the improvement of the

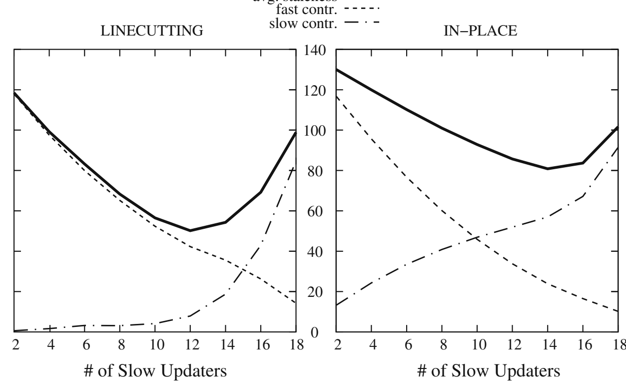


Fig. 9 Slow keys versus fast keys when distribution is varied through n_s

LINECUTTING heuristic had a different trend. Although present in all cases, the improvement picked up and then declined when the ratio between slow and fast keys changed. We can explain this by having a closer look at the contributions that each class of keys had in the overall average staleness. Figure 9 shows this by comparing the contributions of each class of keys plus the average staleness values between LINECUTTING (left graph) and IN-PLACE (right graph). We considered only the case where LF = 15. The contributions for each class were computed as follows: $\text{contrib}_{\text{slow}} = \frac{n_s}{n} \times \overline{\mathscr{S}}_{\text{slow}}$, $\text{contrib}_{\text{fast}} = \frac{n-n_s}{n} \times \overline{\mathscr{S}}_{\text{fast}}$, where $\overline{\mathscr{S}}_{\text{slow}}$ and $\overline{\mathscr{S}}_{\text{fast}}$ are the average staleness values for the corresponding category of keys. This yields the final staleness value to be $\overline{\mathscr{S}} = \text{contrib}_{\text{slow}} + \text{contrib}_{\text{fast}}$.

We can observe that the $\overline{\mathscr{S}}$ value for LINECUTTING followed closely $\text{contrib}_{\text{fast}}$ for slow updaters up to about 10. Then, between 10 and 14, $\overline{\mathscr{S}}$ entered on a diverging slope as contention regions became more frequent, thus making slow updaters compete more and more for being promoted. In the case of IN-PLACE, which makes all keys wait in the queue for the same amount of time, we can observe

a cross-point between the slow and fast contribution values at exactly 10 (i.e., the symmetric distribution). Although the fast keys contributed in a similar manner to the $\overline{\mathscr{S}}$ value as in the case of LINECUTTING, the slow ones had a bigger contribution right from the beginning of the x-axis. This caused a greater staleness value compared with LINECUTTING. We have also considered the same analysis for other load factor values and noticed a similar trend. However, the diverging point for LINECUTTING was seen moving from right to left for increasing levels of load, which would explain the positions of the maximum values for the staleness ratio in Fig. 8b.

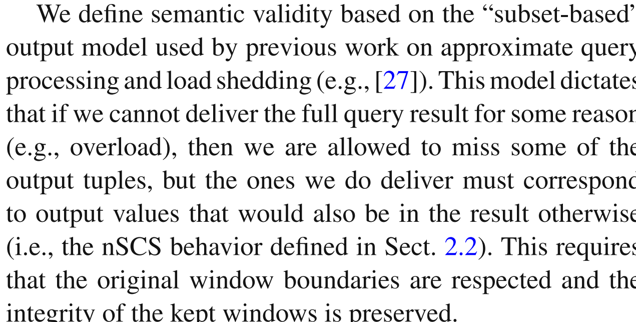
Based on these results, we can draw the following conclusions with respect to the LINECUTTING heuristic. First, the higher the skew, the better the performance. Second, the heuristic responded well to a variety of distribution parameters (n, n_s, σ), making it a great policy to handle the two-class type of update probabilities. Third, the general trend induced by load seemed to be orthogonal to the characteristics of the distribution. However, even at high load, the staleness ratio never went below the improvement threshold. Thus, our experiments validate that LINECUTTING is a better policy than IN-PLACE for non-uniform key frequency distributions with two distinct classes, slow and fast. Based on this positive result, we are planning to investigate a broader spectrum of key frequency distributions as part of our future work, also looking deeper into issues such as optimality guarantees.

5 Handling windows

In this section, we focus on window management in UpStream. Window Manager is concerned with two issues: (i) ensuring the correct way of applying update semantics for sliding window-based query processing, (ii) managing window tuples accordingly. In what follows, we will focus on the logical aspects of our solution for these issues. The physical implementation aspects will be presented in the next section.

5.1 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, resulting 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 units, our problem has an additional challenge compared with the tuple-based scenario: Besides worrying about the recency of our outputs, we must also make sure 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., [27]). This model dictates that if we cannot 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 we do deliver must correspond to output values that would also be in the result otherwise (i.e., the nSCS behavior defined in Sect. 2.2). This requires that the original window boundaries are respected and the integrity of the kept windows is preserved.

Based on the above, we adopt the following two design principles that ensure correct subset-based approximation in UpStream when handling windows:

- 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. We say that we "commit" to processing that window. Changing the decision in the middle of a partially processed window is not allowed.

These two principles help us produce correct outputs. We still need to determine which windows we should commit to, but this decision affects staleness, not correctness. We come back to this issue in Sect. 5.3.

5.2 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 [27]. In this work, load shedding is achieved through a special Window Drop operator that 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. Further details of the Window Drop approach can be found in an earlier publication [27]. Here, we rather focus on how it is adopted in UpStream.

In UpStream, we essentially push the above tuple marking logic down to the storage level. Instead of a Window Drop operator, UpStream uses the update queue to mark stored tuples before the query can process them. As such, our "window-aware" update queues have an advantage over the Window Drop approach: windows that 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 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, to lower staleness, we must keep the most recent windows; therefore, the update queue marks the windows accordingly. There are actually 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.

5.3 Window buffer management

In UpStream, *window buffers* maintain the data structures needed to keep track of window updates. Since updates of different key values are handled independently, there is one window buffer for each update key value. Let us 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 and update semantics is imposed. Two main factors determine how this is maintained: (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 versus 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 designed two alternative window buffer management policies based on whether the commit decision is taken at the window ends (the “lazy buffer”) or at the window starts (the “eager buffer”).

5.3.1 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 10 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 have arrived until b. 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.

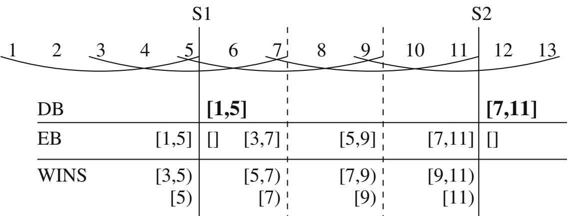


Fig. 10 Lazy window buffer

In order to keep track of all the incomplete windows, the lazy buffer maintains an additional list (WINdowS 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 list and placed in 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. For instance, [1, 5] and [7, 11] are the windows that the buffer manager commits to.

5.3.2 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 (but not 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 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 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.

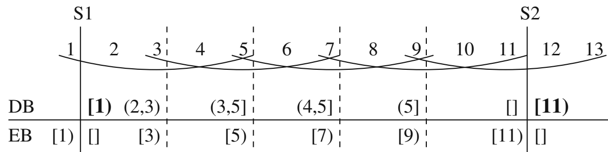


Fig. 11 Eager window buffer

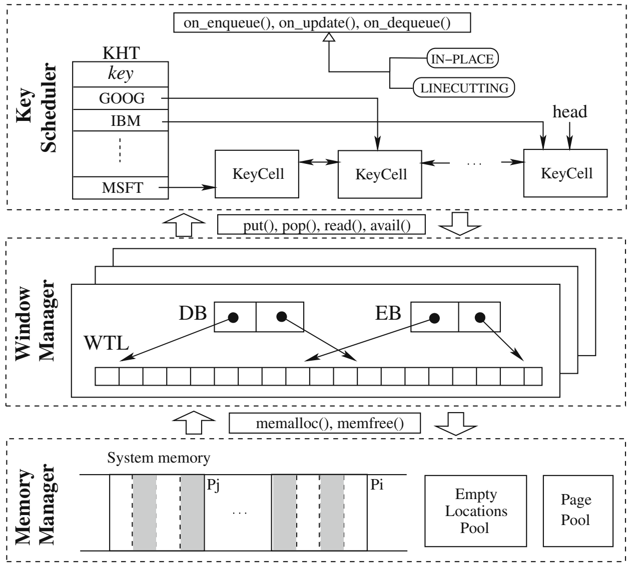


Fig. 12 Update queue manager implementation

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. For instance, the windows starting at 1 ([1)) and 11 ([11)) are the windows that the buffer manager commits to.

6 UpStream implementation

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

As shown in Fig. 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 in which update keys are to be processed; 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.

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. KS is designed to react to the main events related to the queue of keys: Update for a key that is not in the queue (on_enqueue()), update for a key that is waiting to be serviced (on_update()) and removing a key for processing (on_dequeue()). These events are handled differently according to each key scheduling policy. For instance, IN-PLACE adds a new key at the end of the queue, does nothing when keys in the queue update, and simply removes the head at dequeue. In this case, keeping the queue of keys sorted by waiting time comes for free. LINECUTTING, on the other hand, would have to consider update frequencies (retrieved from the Statistics Monitor) in order to keep the queue of keys ordered using both slowness and waiting time.

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 WM. 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. A Window Tuple List (WTL) in turn holds pointers to the actual memory locations where the tuples that belong to those windows are physically stored. KS communicates with the window buffer in WM through a typical sequential storage interface (pop(), put(), read(), and avail()).

Finally, the physical memory locations are handled by the MM layer. WM communicates with MM through the memalloc() and memfree() methods, in order to make 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 (EPL) indexed by pages. When all the memory locations within a page are freed, the page is handed back to the Page Pool. When EPL is empty, a new page needs to be allocated from the Page Pool. For further details about our implementation, we refer the readers to our technical report [18].

7 Performance

In this section, we provide an experimental evaluation of the storage-based load management techniques that we have proposed in this paper. The first goal of our experimental study is to prove the efficiency of our storage-centric approach in adapting to overloading conditions and lowering staleness.

Our second goal is to explore the performance space of staleness by analyzing the effects of both different key scheduling techniques and windowing semantics.

7.1 Experimental setup

System: We implemented our UpStream framework as part of the Borealis stream processing system, expanding on its storage manager component [1]. 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 an update-at-a-time basis, using the "super-box" scheduling technique [9] 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 used a single-node setup for Borealis running on a Linux box with an Intel Quad Core Intel Xeon 3360 2.8 GHz processor and 8 GB of memory.

Workload: In our experiments, we used both synthetic and realistic data sets. Synthetically generated streams contain tuples of the form (*time, update key, value*). The input is ordered by the *time* field. The number of distinct *update key* values ranged between 1 and 100, depending on the experiment. The actual values of the *value* field do not have any significance in terms of what we measure in the experiments; thus, they were randomly chosen from a numeric domain. We also used data streams generated based on update rate distributions taken from a real use case: the NYSE TAQ data from 2006 [20]. 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 used a single query, whose operators were 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 used the "delay" operator of Borealis. 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 of representing a query plan with a certain CPU cost; its delay parameter provides a knob to easily adjust the query cost (measured in milliseconds). In the tuple-based scenarios, using one delay operator was sufficient for the purpose, whereas in window-based scenarios, we also used a sliding window aggregate operator that was placed either upstream or downstream from a delay operator. This helped us explore whether 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 investigated two performance aspects in our experiments:

- Staleness: As formulated in Sect. 2.3, per-key average staleness was computed over individual staleness values of all output tuples generated for each distinct update key value across a given run period. Then, an overall average was computed across all the keys. Staleness was measured in seconds.
- Memory usage: Maximum memory allocated from the UpStream Page Pool was recorded between consecutive output deliveries. Then, we computed a maximum over all these recordings to see the worst-case memory usage for UpStream across a given run. Memory usage was measured in the number of stored tuples.

We also report on the accuracy of the queries under subset-based query approximation. Recall that in the simulator, the load factor (LF) was a controlled variable that directly modeled load to the system. In fact, LF also models the selectivity of the query. For instance, a value of LF = 10 means that out of 10 results that could have been produced, only 1 was delivered and 9 discarded. In a real stream processing system, LF becomes an observed variable. We will report the accuracy of the queries using exactly this measure of load.

Note that the results presented were obtained based on averaging over repeated runs in order 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 and drop probability in random drop scenarios); (ii) the randomness in the order of certain system events occurring beyond our control (such as how the enqueue thread and the scheduler thread in Borealis are synchronized by the operating system).

7.2 Key scheduling

The following set of results focuses on evaluating our key scheduling strategies on tuple-based queries. First, we look at the improvement brought by in-place updates versus random drops when update keys update at the same rates (Sect. 7.2.1). We then investigate the benefits of the LINECUTTING key scheduling strategy when update rates are non-uniformly distributed (Sect. 7.2.2).

7.2.1 In-place updates versus random drops

We ran the scenarios in Table 1 with an input stream consisting of multiple update key values. Scenario (a) represents the state-of-the-art in terms of load shedding using random

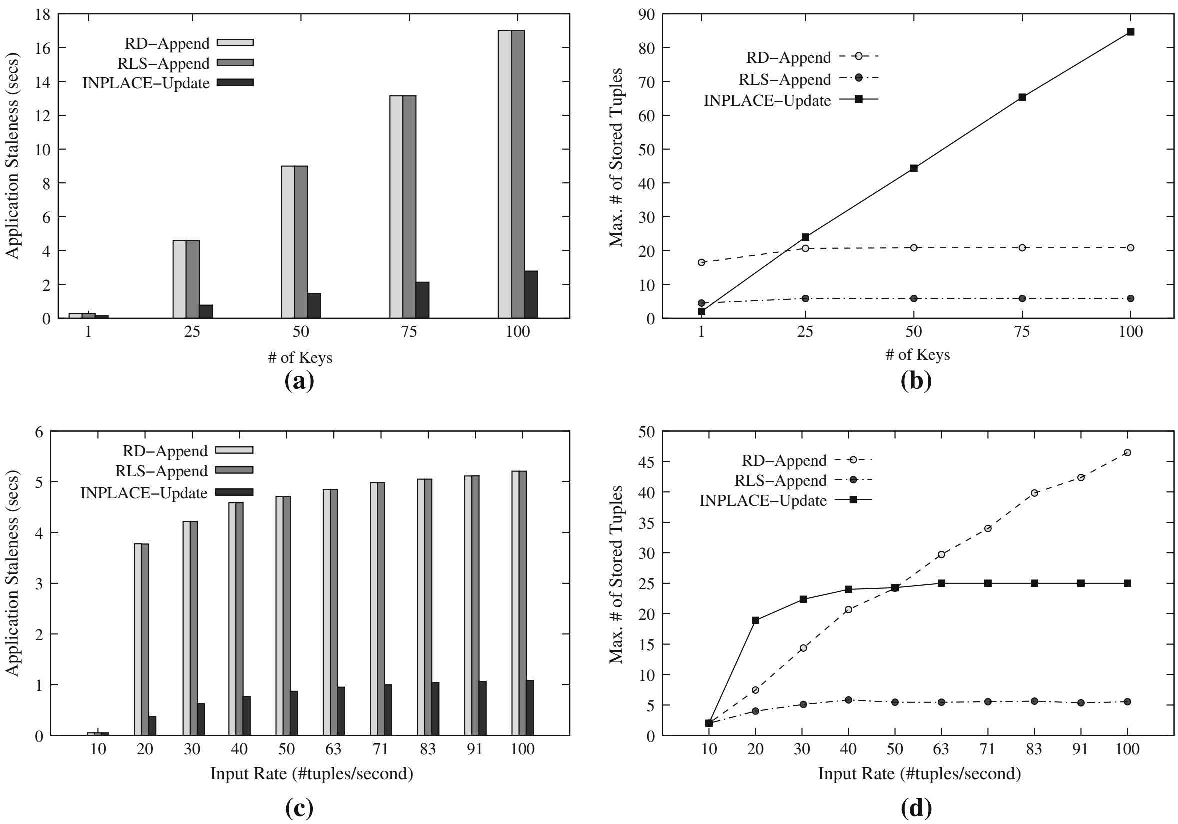


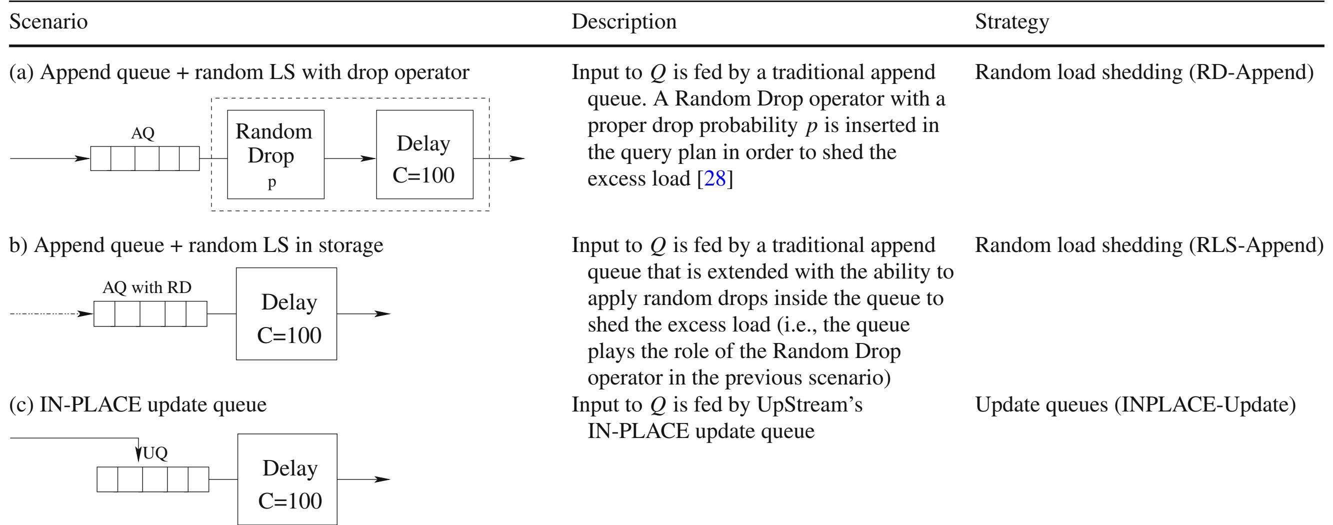
Fig. 13 Update queue versus random load shedding variants (multiple update keys) **a** Staleness versus # of keys. **b** Memory usage versus # of keys. **c** Staleness versus input rate. **d** Memory usage versus input rate

drops, (b) is an adaptation of (a) to match a storage-centric approach, and (c) is UpStream. We measured staleness and memory usage for an increasing number of update keys as well as increasing input rates in the system. Our results are shown in Fig. 13.

First, we varied the number of keys between 1 and 100 (x-axis), while fixing the input rate to 40 tuples/s. 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 used a ZipF distribution with skew parameter $\theta = 0$ [14] and with a different seed value at every repeated run. The results of this experiment are shown in Fig. 13a and b. All three compared approaches exhibit increasing staleness with increasing number of keys. Staleness grew for the INPLACE-Update scenario, because the average length of the update queue grows when more keys update uniformly at the same time. In this case, all keys waited in the queue for nearly the same amount of minimum possible time, which was directly correlated with the number of distinct update keys. The load shedding scenarios, however, did 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 at random across different key values. The effect of this on staleness becomes even more apparent when the number of keys increases. The trend for memory usage, on the other hand, is somewhat the opposite. As in the single-key experiment, the load shedding approaches consumed fixed memory space, although the storage-based variant was slightly more efficient than the operator-based variant. However, the update queue's memory usage was 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 set the number of keys to 25 and varied the input rates so as to obtain a corresponding load factor that varied linearly between 1 and 9. The results of this experiment are shown in Fig. 13c and d. The memory usage of our update queue was correlated with the number of update keys (in this case, 25), which required more memory slots to be allocated. This was useful in order to keep the latest update for all keys that were in the queue at any time and effectively lowering staleness. On the other hand, the RLS-Append approach shows lower memory consumption at the price of higher staleness levels. This is because tuples are randomly dropped

Table 1 Tuple-based processing scenarios



Scenario	Description	Strategy
(a) Append queue + random LS with drop operator AQ → Random Drop p → Delay C=100 →	Input to Q is fed by a traditional append queue. A Random Drop operator with a proper drop probability p is inserted in the query plan in order to shed the excess load [28]	Random load shedding (RD-Append)
b) Append queue + random LS in storage AQ with RD → Delay C=100 →	Input to Q is fed by a traditional append queue that 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)	Random load shedding (RLS-Append)
(c) IN-PLACE update queue UQ → Delay C=100 →	Input to Q is fed by UpStream's IN-PLACE update queue	Update queues (INPLACE-Update)

as they arrive in the queue. For RD-Append, the penalty is paid for postponing the drop process until the Random Drop operator gets the chance to see the tuples. In this case, tuples tend to accumulate more in the queue as load grows.

7.2.2 LINECUTTING versus IN-PLACE

Section 4.3 described the LINECUTTING key scheduling strategy as a way to exploit non-uniform key update frequencies. LINECUTTING showed good potential for reducing staleness as we evaluated it against IN-PLACE in the simulator using our study distribution with slow and fast updaters. Here, we show the same comparison but in the experimental setup offered by a real stream processing system. We present the results of this comparison based on running three workloads: two synthetic ones and one taken from a real-life use case. The data streams were run in the context of scenario (c) in Table 1 with an input rate of 1000 tuples per second and a delay in producing a result of 5 ms. Our goal is to observe the effects of skew and number of updating keys.

Slow versus fast

We start with a verification of the LINECUTTING benefits while running queries on a stream containing slow and fast updaters. Similar to the setup described in Fig. 8a, we observed the effect of different skew settings between the two classes of updaters. In addition, the number of total updating keys in the stream was varied as shown in Fig. 14. We measured the percent improvement between LINECUTTING and IN-PLACE.

The first point to make is that these results confirm the overall trend of increasing benefits with increasing skew. Secondly, we expected LINECUTTING to improve more on

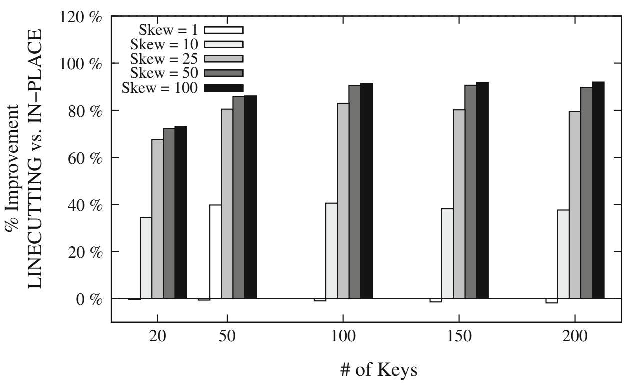


Fig. 14 Comparison of LINECUTTING and IN-PLACE for update rate distributions with fast and slow updaters

staleness as the number of keys increased. This trend should have been caused by the number of slow updaters growing proportionally. At high skew (e.g., 100), our expectations were indeed met. However, the trend differed slightly with a smaller skew. Our prime suspect is the overhead introduced by LINECUTTING due to its decision process at dequeue time. Although we argued in Sect. 4.3 that LINECUTTING has a fail-safe mechanism that prevents degradation compared with IN-PLACE by considering the wait time W, we can observe some small degradation at skew = 1, i.e., no skew. Moreover, the improvement slightly degrades with the number of keys for the same skew setting. At this point, the fail-safe seems to have been slightly overrun by LINECUTTING's own overhead. However, overall, LINECUTTING shows clear improvement over IN-PLACE at all settings.

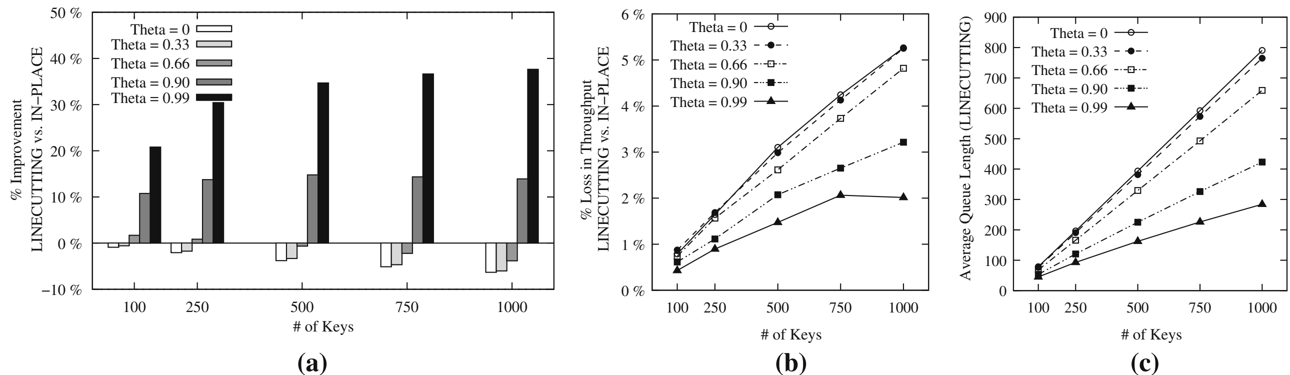


Fig. 15 Comparison of LINECUTTING and IN-PLACE when the update keys follow a ZipF distribution of update rates. **a** Improvement in staleness exhibited by LINECUTTING relative to IN-PLACE. **b** Loss in throughput observed for LINECUTTING relative to the one observed for IN-PLACE. **c** Average queue length achieved by LINECUTTING

Generalized benefits and overhead analysis

Next, we expanded our analysis to a more generic update distribution to show that the benefits of LINECUTTING are not bound to our studied distribution. To that extent, we considered a stream of keys whose update rates followed a ZipF distribution. This offered us a simple way to vary the workload with the help of the skew parameter θ, which takes values from the [0,1) range: The value 0 indicates no skew between the update keys, while approaching 1 increases the skew. We varied our workload by considering both the θ parameter and the number of keys, similar to our previous experiment in Fig. 14. In this case, the x-axis contains the number of keys used in the ZipF distributions (see Fig. 15b). On the skewness front, LINECUTTING showed definite improvement only toward the end of the θ range whereas degradation was installed in the first part (0–0.66). With increasing numbers of keys, higher skew caused better performance, whereas lower skew caused increasing degradation. This trend was also confirmed in the case of the study distribution.

Figure 15c and d help us explain the source of degradation. Using the same x-axis as in Fig. 15b, we show the loss in throughput exhibited by LINECUTTING relative to IN-PLACE. Given the same runtime, we compute throughput as the number of results per total duration of the experiment (i.e., number of results per second). Normally, under load shedding conditions, the throughput indicates the capacity of the system. However, any kind of overhead added to the processing time can decrease the capacity. As seen in Fig. 15c, LINECUTTING does show loss in throughput which, as expected, grows with the number of keys, and is reduced by higher skew. The correlation to the graph in Fig. 15d suggests the cause. The graph shows the average queue length registered by LINECUTTING. By traversing the entire queue to reach a decision, LINECUTTING introduces overhead that grows with the queue length.

Case study: monitoring financial data

Finally, let us have a look at a realistic distribution. We considered the data set containing the trades and quotes from a day in January 2006 registered at NYSE [20]. Given the observed update rates for stock symbols, we extracted their update probabilities. Based on the entire range of symbols, we devised several workloads as follows. Imagine the user of a financial application monitoring market data. The user is interested in a certain amount of stock symbols that form his or her stock portfolio. Our workloads consisted of stock portfolios with different numbers of monitored symbols selected uniformly across the entire range, that is, to make up a diversified portfolio.

We ran the streams made up of symbols for each portfolio through our system first configured with an IN-PLACE update queue and then with a LINECUTTING one. The results are shown in Fig. 16 where the x-axis represents the size of the portfolios and the y-axis measures the application staleness. For all considered workloads, LINECUTTING improved on IN-PLACE, indicating that all portfolios

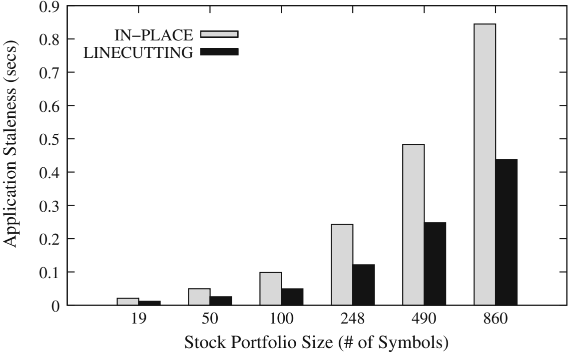


Fig. 16 Staleness for LINECUTTING and IN-PLACE for distributions taken from NYSE financial data

exhibited substantial skew. For instance, the last measurement yielded a value of 0.8 s in staleness for IN-PLACE, whereas LINECUTTING managed to reduce it to almost half of that. We did also observe loss in throughput with a maximum of 3% for the last point. However, the skew seems to have been high enough that the benefits of LINECUTTING exceeded its overhead by far: the improvement in staleness was reported earlier 70% for all points.

7.3 Window-based processing

In the next set of experiments, we used different configuration scenarios (see Table 2) of a window-based query Q, consisting of a sliding window aggregate operator with window size w and slide s and a delay operator with cost C.

We first wanted to observe the benefit of using a window-aware update queue (Sect. 7.3.1), in terms of both staleness and memory usage over the state-of-the-art. Then, we focused on the window buffer management aspect of our window-aware update queues (Sect. 7.3.2) and performed an in-depth evaluation of how the eager and lazy techniques perform under different settings, first in terms of staleness and then memory usage. Hereinafter, our windowing techniques will be denoted by EAGER-WB and LAZY-WB.

7.3.1 Using a window-aware update queue

We ran scenarios (a), (b), and (d) in Table 2 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. These settings created a similar load management scenario as in the Window Drop-based scenario making the results more directly comparable. Window drop decisions are made for the whole window, which is similar to the lazy approach. In this experiment, we used a simple tumbling window with $w = s = 10$ and set Window Drop's batch size $B = 10$. These settings created an ideal scenario for the window drop to be the most effective. This way, we could observe the improvement (if any) of our techniques, even in such an ideal setting for the state-of-the-art.

Our results are shown in Fig. 17. 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, as illustrated by the behavior of the append queue (note the log-scale on the y-axis). Using a Window Drop operator (WDrop scenario) certainly relieved 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 was made without considering the recency of the windows. Therefore, as in the tuple-based processing case, staleness was still higher compared with our window-aware update queue. An additional interesting result we see on the graph of Fig. 17a is that, for WDrop, 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. This is also confirmed by the Load Factor levels (Table 17c) achieved in the WDrop scenario, which get increasingly higher than those in the case of W-Append and LAZY-WB. Finally, on the memory usage front, the result that we depict in Fig. 17b shows once again that doing random drops using an operator has negative effects on memory consumption in the input queues.

7.3.2 Window buffer management: eager versus lazy

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

Single update key

We ran the scenarios (c) and (d) from Table 2 with EAGER-WB or LAZY-WB, respectively, and measured how their staleness changed with increasing load. The window size w was set to 10, but the window slide s was varied as 1, 5, and 10. Please note that the window slide introduces an additional way to increase load (Fig. 18c–e shows that for the same input rate, the accuracy loss varies with window slide as load increases).

The results are shown in Fig. 18a and b. We present two pairs of graphs: the one in Fig. 18a focuses on CPT-style processing on the scenario (c) and the one in Fig. 18b focuses on CPW-style processing on the scenario (d). Overall, we see that in both pairs of 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 created variations in staleness behavior, which we will further explain later. However, it is important to note at this point that, by looking at these two graphs side-by-side, we can immediately observe that EAGER-WB performed better than LAZY-WB when the delay operator was in front of the windowing operator (CPT) and vice versa when the delay was applied after the windowing operator (CPW). Therefore, we will explain our results around a detailed analysis of the CPT and CPW scenarios.

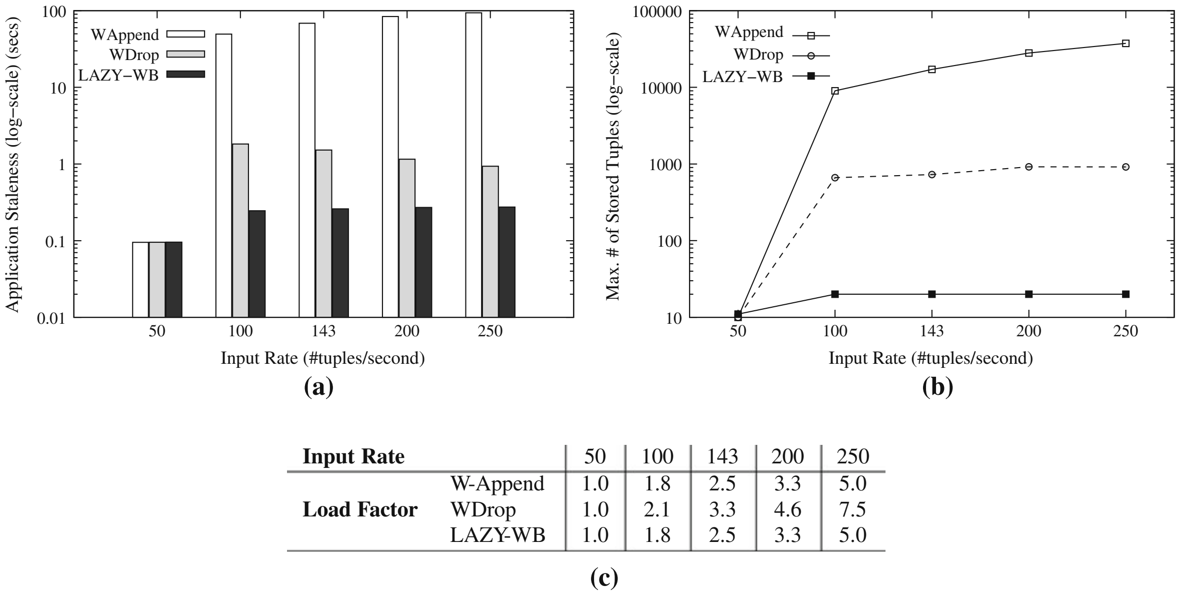
The CPT Case: In Fig. 18a, the staleness achieved by EAGER-WB and LAZY-WB stabilizes toward the same level with increasing load. We will further explain this matter using Table 3 that shows how EAGER-WB and LAZY-WB should behave in time in the CPT and CPW scenarios. Let us focus on the first row. At the top of the figures, we depicted the time spent by the query producing an output after committing to one window. At the bottom, one can see window

Table 2 Window-based processing scenarios

Scenario	Description	Strategy
(a) Append queue	Input to Q is fed by a traditional append queue with FIFO ordering (i.e., no load is shed)	None (WAppend)
(b) Append queue + random LS with Window Drop operator	Input to Q is fed by a traditional append queue. A Window Drop operator with a proper drop probability p and batch size B is inserted in the query plan in order to shed the excess load [27]	Window drop (WDrop)
(c) IN-PLACE update queue + downstream windowing (CPT)	Input to Q is fed by UpStream's window-aware IN-PLACE update queue. 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 assume an "incremental" window processing mode, in which the aggregate function is applied incrementally as each tuple of a given window arrives.	Window-aware update queues (EAGER-WB/LAZY-WB)
(d) IN-PLACE update queue + upstream windowing (CPW)	The input to Q is fed by UpStream's window-aware IN-PLACE update queue. The order of the aggregate and delay operators is switched. 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 assume 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)	Window-aware update queues (EAGER-WB/LAZY-WB)

formation over the input stream as considered by the window buffer. A window should take $w \times TBA$ time units to arrive and $w \times CPT$ to be processed by the query. 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. However, x has different values depending on the window buffer type. For instance, LAZY-WB does not let the query start processing until it has a fully formed window, yielding $x = w \times CPT$. EAGER-WB, on the other hand, exhibits a wider variation for x. The general formula would be $x = w \times CPT - (w - 1) \times TBA$. The lower bound for x is found when load is at system capacity ($CPT = TBA$) 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 $(LF = \frac{CPT}{TBA}) : x = TBA + w \times CPT \times (1 - \frac{1}{LF})$. For very high LF values, the upper bound for x tends to approach $w \times CPT$, which is also the value achieved by LAZY-WB. This is confirmed by the results shown in Fig. 18a, where we can see that for the first half of the x-axis, EAGER-WB achieved lower staleness than LAZY-WB.

The CPW Case: In the CPW scenario (see Table 3, second row), a window result waits 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



Input Rate		50	100	143	200	250
Load Factor	W-Append	1.0	1.8	2.5	3.3	5.0
	WDrop	1.0	2.1	3.3	4.6	7.5
	LAZY-WB	1.0	1.8	2.5	3.3	5.0

(c)

Fig. 17 Window-aware update queue versus append queue variants (CPW, LAZY-WB, single update key). **a** Staleness versus input rate. **b** Memory usage versus input rate. **c** Observed load factor

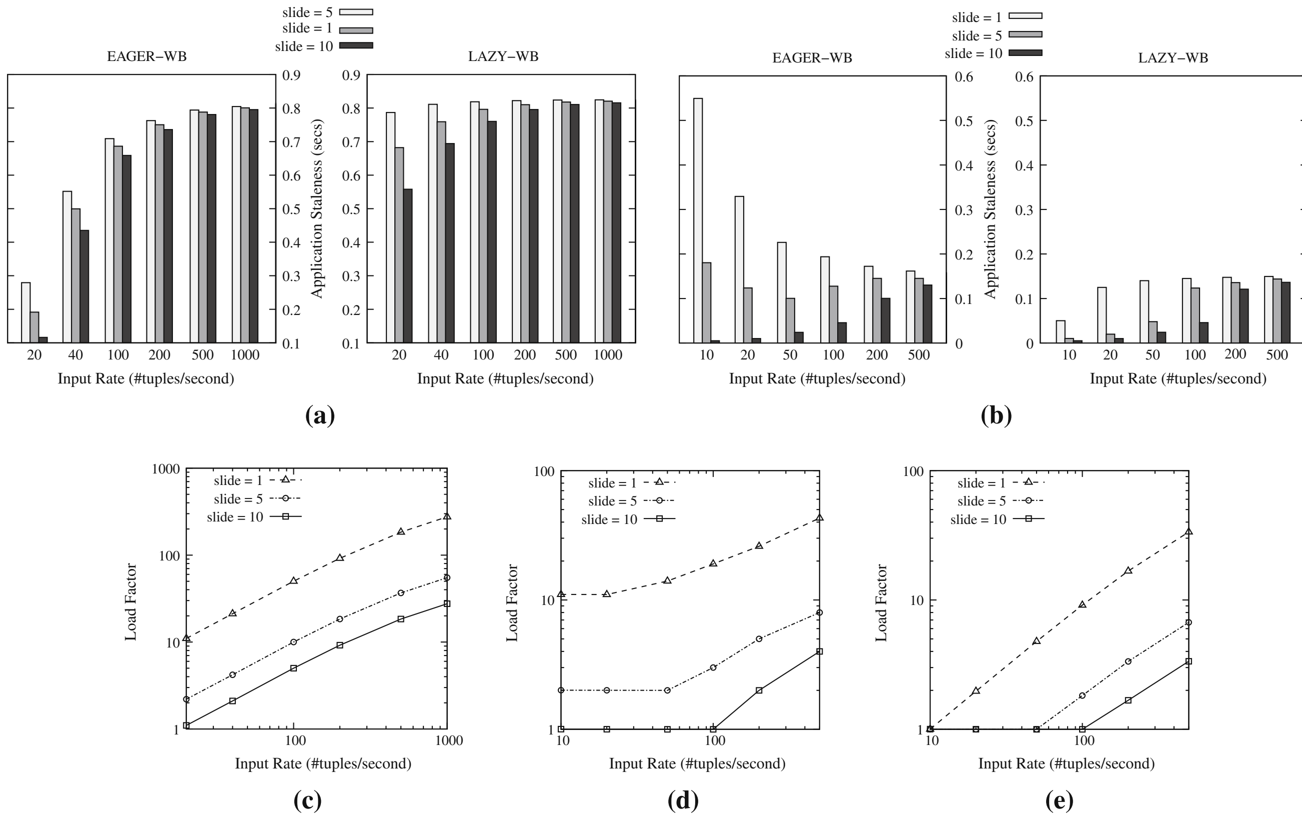


Fig. 18 Eager versus lazy window buffer management on window-aware update queues (single update key). **a** CPT on delay + windowing. **b** CPW on windowing + delay. **c** Observed load factor for CPT. **d** Observed load factor for CPW, EAGER-WB. **e** Observed load factor for CPW, LAZY-WB

show that EAGER-WB behaved worse than LAZY-WB for the first half of the considered load spectrum. For the second half, where the load was very high, the two techniques stabilized 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,

Table 3 Window Buffer behavior for CPT and CPW (TBA = Time Between Updates, CPT = Cost Per Tuple, CPW = Cost Per Window)

	EAGER-WB	LAZY-WB
CPT	Query: w*CPT, w*CPT; Window Buffer: w*TBA, x, x	Query: w*CPT, w*CPT; Window Buffer: w*TBA, x, x
CPW	Query: y, CPW, y, CPW; Window Buffer: w*TBA, x, x	Query: y, CPW, y, CPW; Window Buffer: w*TBA, x, x

On the Window Buffer side, the black rectangles are the most recent window updates while the empty ones represent ignored windows. On the Query side, the gray-filled rectangles represent the time spent by the query to dequeue/process a window and report a result. The latter is marked by an arrow pointing up

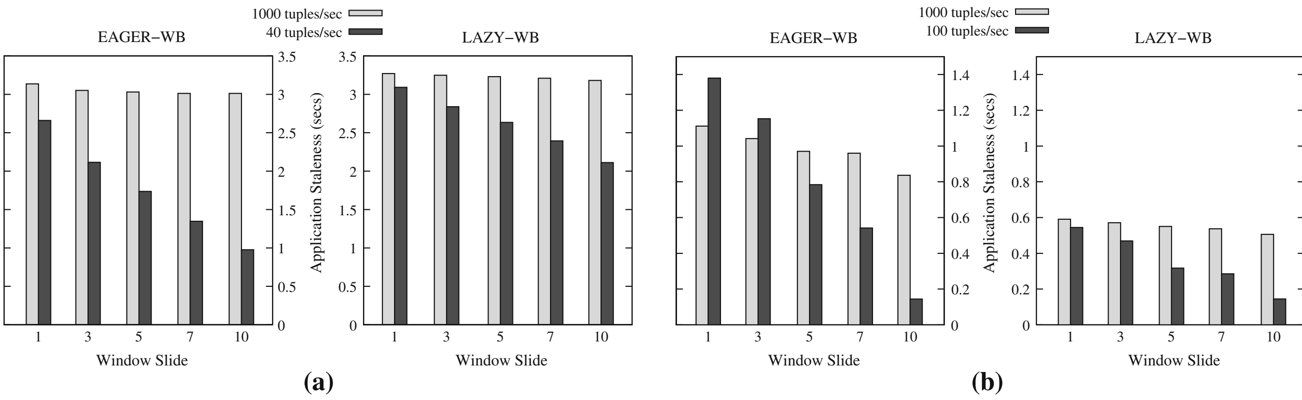


Fig. 19 Staleness for eager versus lazy window buffer management on window-aware update queues (multiple update keys). **a** CPT. **b** CPW

y is negligible compared with the cost of the downstream query (i.e., CPW), since the window is already in the storage and the operator can consume it very fast. By contrast, in the case of EAGER-WB, the aggregate operator dequeues and computes window tuples as they arrive. In this case, y can get to the maximum of $w \times TBA = w \times \frac{CPW}{LF}$. This is only reached in the non-overload scenario, explaining the strange maximum that was achieved by EAGER-WB at $LF = 1$. As LF increased, y became smaller and smaller, causing the staleness achieved by EAGER-WB approach that achieved by LAZY-WB for high load levels.

Multiple update keys

Next, we repeated the previous experiment for multiple update keys. For this experiment, we set the number of update keys to 10, 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 fast sliding windows that highly overlap and lead to a higher number of windows to be constructed and processed. We took measurements for EAGER-WB and LAZY-WB for two types of input rates, low (40 or 100 tuples/s) and high (1000 tuples/s).

Figure 19 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 expected to observe in the multi-key scenario is that windows of a given key may experience time gaps due to tuple arrivals for other keys. We expected 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 these experiments through a detailed analysis around the CPT and CPW scenarios.

The CPT Case: In Fig. 19a, at very high load, EAGER-WB and LAZY-WB are closer in terms of staleness. We also measured the average length of the queue for the entire run time; for the high load setting (1000 tuples/s), the reported values

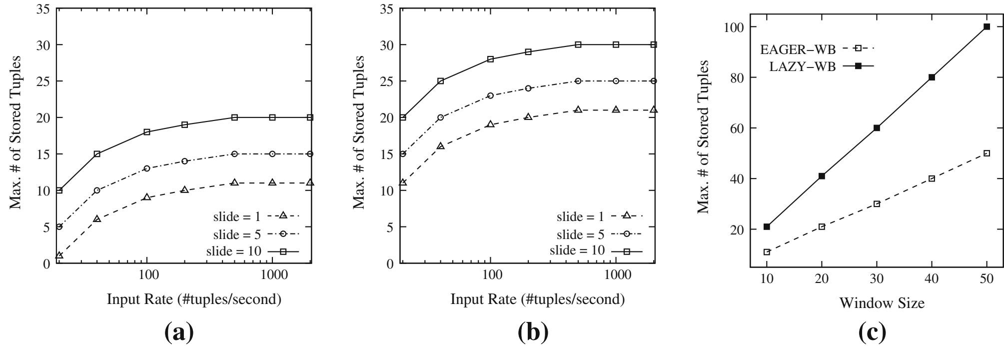


Fig. 20 Memory usage for eager versus lazy window buffer management on window-aware update queues. **a** Memory usage versus input rate (EAGER-WB). **b** Memory usage versus input rate (LAZY-WB). **c** Memory usage versus window size

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 slightly better than LAZY-WB. The reason is that it takes advantage of gaps in the windows by immediately starting to serve other keys when a gap is hit. In this case, referring to Table 3, x is equal to $n_c \times CPT$, where n_c is the number of keys that have closed their committed windows before this key closed its own committed window. For high load levels, gaps closed faster compared with CPT, making n_c smaller as load increased. In this case, staleness for EAGER-WB was mostly affected by the number of keys in the queue. LAZY-WB exhibits the same trend for x as in the single-key scenario ($w \times CPT$), and hence, staleness for LAZY-WB was mostly due to the average queue length. It appears that for high load, both buffers tend to level staleness off. By contrast, we can observe EAGER-WB doing better than LAZY-WB for lower load levels. This is because EAGER-WB was able to benefit more from gaps and interleaving the processing of keys, yielding lower values for n_c.

The CPW Case: In Fig. 19b, we see EAGER-WB being affected by gaps in a rather negative way. If we place Table 3 (second row) in the current context, the only thing that changes is y. This is where gaps came into play. y is the time spent by the aggregate to dequeue and compute a window for a key. In this case, since window tuples took longer to arrive, y also included the accumulated length of the gaps, disfavoring EAGER-WB at smaller window slide values.

Memory usage

In this last experiment, we wanted 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 has more influence on memory usage than CPW. We expected EAGER-WB and LAZY-WB to exhibit different memory usage patterns due to the difference in the way windows are overwritten. We expected that the amount of consumed memory would also depend on the window size, window slide, and the system load.

The first two graphs (Fig. 20a, b) show 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 $2 \times w + s$, respectively. We can see from the graphs that these bounds were reached at very high system loads, where memory usage stabilized. Our techniques achieve bounded memory usage.

The third graph (Fig. 20c) shows memory usage for different window sizes. The input rate was set to 1000 tuples/s and the window slide was set to $s = 1$ (i.e., high load, extreme degree of window overlap). It is clear to see 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. The results of this experiment indicate once again that EAGER-WB requires less memory than LAZY-WB.

7.4 Summary

We conclude this performance evaluation section with a summary of our findings.

Our experiments confirmed that our storage-centric approach is efficient in *adapting to overload*, *lowering staleness*, and *controlling memory consumption*, when we compared UpStream against state-of-the-art in terms of load shedding using random drops (for both tuple-based and window-based operations). Moreover, UpStream can directly minimize and control staleness via *efficient key scheduling*. We also verified that the IN-PLACE strategy can minimize staleness when keys update uniformly both in theory and in practice. The LINECUTTING scheduling strategy on the other hand showed definite improvement compared with IN-PLACE, when considering non-uniform key update rate distributions (both synthetic and realistic), although in some cases the overhead of the decision making process slightly degraded the benefits.

We also observed the performance benefits and trade-offs of using two different window management strategies for our update queues: eager and lazy. Although they both showed similar staleness levels at very high system load, we saw cases where eager performed better, i.e., *incremental window processing*, and others where lazy was more suitable, i.e., *non-incremental window processing*. In terms of memory consumption, both strategies were able to achieve bounded memory usage.

8 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., [32]) and dynamically changing this placement as data arrival rates change (e.g., [6,22,31]). 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 that result in approximate answers (e.g., [5,25,27–30]). 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 [10] study the problem of update synchronization of local copies of remote database sources in a web environment. 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. More recent work by Qu et al. [23,24] considered two types of transactions for web databases: query transactions and update transactions. 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 represented by 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. [26] propose a scheduling policy for multiple continuous queries so as to maximize the freshness of the output streams disseminated by a web server. 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 is relevant to our work as well. The STanford Real-time Information Processor (STRIP) separates view imports from view exports. In this model, both the base data and the derived data materialized as views must be refreshed as updates are received (view import). Also, read transactions on both the base data and materialized views must be executed, with specific deadlines (view export). As in web databases, this creates a trade-off between response time for read transactions and freshness for update transactions. Adelberg et al. propose scheduling algorithms for efficient updates on base data [2], as well as on derived data [3]. Kao et al. [15] further extend these works by proposing scheduling policies for ensuring temporal consistency. A more recent and closely related work to UpStream is the DataDepot Project from AT&T Labs [13,12,7]. DataDepot is a tool for generating data warehouses from streaming data feeds, and therefore, it has many data warehousing 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). The two projects are complementary since UpStream can potentially serve as a preprocessor for a real-time data warehouse system such as DataDepot.

9 Conclusions and future work

In this paper, 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 and space-efficient window processing techniques that ensure correct and low-staleness results for sliding window queries. We would like to address the following issues as part of our future work:

- We showed in this paper that the update queue can minimize staleness when using the IN-PLACE policy, and the update keys are uniformly distributed in terms of their frequencies. Furthermore, based on our LINECUTTING heuristic, we have shown that better key scheduling algorithms can be devised for the non-uniform case. As part of our future work, we would like to extend our UpStream framework to include additional key scheduling policies that may be a better fit for a broader set of update key frequency distributions.
- In this paper, 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 for all update keys at the end point application. However, the application may also want to access the results at different rates (e.g., GOOG stocks to refresh every minute and GM stocks to refresh every hour). Therefore, we intend to integrate application-specific access frequencies into our QoS model.
- Lastly, our storage framework allows append and update queues to coexist 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.

Acknowledgments We would like to thank Gustavo Alonso, Donald Kossmann, and Timothy Roscoe for their valuable feedback on earlier versions of this paper and Simonetta Zysset for proofreading the final manuscript. The work presented in this paper has been supported in part by the National Competence Center in Research on Mobile Information and Communication Systems NCCR-MICS, a center supported by the Swiss National Science Foundation under grant number 51NF40-130758/1.

References

1. Abadi, D., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the Borealis stream processing engine. In: CIDR Conference, Asilomar, CA (2005)
2. Adelberg, B., Garcia-Molina, H., Kao, B.: Applying update streams in a soft real-time database system. In: ACM SIGMOD Conference. San Jose, CA (1995)
3. Adelberg, B., Kao, B., Garcia-Molina, H.: Database support for efficiently maintaining derived data. In: EDBT Conference. Avignon, France (1996)
4. Alonso, R., Barbara, D., Garcia-Molina, H.: Data caching issues in an information retrieval system. ACM Trans. Database Syst. **15**(3), 359–384 (1990)
5. Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams. In: IEEE ICDE Conference. Boston, MA (2004)
6. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Contract-based load management in federated distributed systems. In: NSDI Conference. San Fransisco, CA (2004)
7. Bateni, M.H., Golab, L., Hajiaghayi, M.T., Karloff, H.: Scheduling to Minimize Staleness and Stretch in Real-Time Data Warehouses. In: ACM SPAA Conference. Calgary, Canada (2009)
8. Botan, I., Alonso, G., Fischer, P.M., Kossmann, D., Tatbul, N.: Flexible and scalable storage management for data-intensive stream processing. In: EDBT Conference. Saint Petersburg, Russia (2009)
9. Carney, D., Çetintemel, U., Rasin, A., Zdonik, S.B., Cherniack, M., Stonebraker, M.: Operator scheduling in a data stream manager. In: VLDB Conference. Berlin, Germany (2003)
10. Cho, J., Garcia-Molina, H.: Synchronizing a database to improve freshness. In: ACM SIGMOD Conference. Dallas, TX (2000)
11. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In: VLDB Conference. Rome, Italy (2001)
12. Golab, L., Johnson, T., Seidel, J.S., Shkapenyuk, V.: Stream warehousing with DataDepot. In: ACM SIGMOD Conference. Providence, RI (2009a)
13. Golab, L., Johnson, T., Shkapenyuk, V.: Scheduling updates in a real-time stream warehouse. In: IEEE ICDE Conference. Shanghai, China (2009b)
14. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: ACM SIGMOD Conference. Minneapolis, MN (1994)
15. Kao, B., yiu Lam, K., Adelberg, B., Cheng, R., Lee, T.S.H.: Updates and view maintenance in soft real-time database systems. In: CIKM Conference. Kansas City, MO (1999)
16. Labrinidis, A., Roussopoulos, N.: Exploring the tradeoff between performance and data freshness in database-driven web servers. VLDB J. **13**(3), 240–255 (2004)
17. Maskey, A., Cherniack, M.: Replay-based approaches to revision processing in stream query engines. In: SSPS Workshop. Nantes, France (2008)
18. Moga, A., Botan, I., Tatbul, N.: UpStream: storage-centric load management for data streams with update semantics. Tech. Rep. Technical Report TR-620, ETH Zurich Department of Computer Science (2009) ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/620.pdf
19. Muthukrishnan, S.: Data streams: Algorithms and applications. Foundations and Trends in Theoretical Computer Science 1(2) (2005)
20. NYSE (2006) NYSE Data Solutions. http://www.nyxdata.com/nysedata/
21. Olston, C., Widom, J.: Best-Effort Cache Synchronization with Source Cooperation. In: ACM SIGMOD Conference. Madison, WI (2002)
22. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: IEEE ICDE Conference. Atlanta, GA (2006)
23. Qu, H., Labrinidis, A.: Preference-aware query and update scheduling in web-databases. In: IEEE ICDE Conference. Istanbul, Turkey (2007)

24. Qu, H., Labrinidis, A., Mosse, D.: UNIT: user-centric transaction management in web-database systems. In: IEEE ICDE Conference. Atlanta, GA (2006)
25. Reiss, F., Hellerstein, J.M.: Data triage: an adaptive architecture for load shedding in TelegraphCQ. In: IEEE ICDE Conference. Tokyo, Japan (2005)
26. Sharaf, M.A., Labrinidis, A., Chrysanthis, P.K., Pruhs, K.: Freshness-aware scheduling of continuous queries in the dynamic web. In: WebDB Workshop. Baltimore, MD (2005)
27. Tatbul, N., Zdonik, S.: Window-aware load shedding for aggregation queries over data streams. In: VLDB Conference. Seoul, Korea (2006)
28. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: VLDB Conference. Berlin, Germany (2003)
29. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying FIT: efficient load shedding techniques for distributed stream processing. In: VLDB Conference. Vienna, Austria (2007)
30. Tu, Y., Liu, S., Prabhakar, S., Yao, B.: Load shedding in stream databases: a control-based approach. In: VLDB Conference. Seoul, Korea (2006)
31. Xing, Y., Zdonik, S., Hwang, J.H.: Dynamic load distribution in the Borealis stream processor. In: IEEE ICDE Conference. Tokyo, Japan (2005)
32. Xing, Y., Hwang, J.H., Çetintemel, U., Zdonik, S.: Providing resiliency to load variations in distributed stream processing. In: VLDB Conference. Seoul, Korea (2006)