UpStream: Storage-centric Load Management for Data Stream Processing Systems

Alexandru Moga
(supervised by Nesime Tatbul)
Systems Group, Computer Science Department
ETH Zurich, Switzerland
amoga@inf.ethz.ch

ABSTRACT

Processing fast updating data streams in real-time must reflect the most recent data. A number of technologies including Data Stream Management Systems have emerged to respond to this challenge. While running their queries in a continuous fashion on high-volume push-based data streams (e.g. sensor data, GPS coordinates, stock quotes), one of the most important optimization problems that these systems face is load management based on Quality of Service (QoS). In stream processing, QoS may depend on a number of factors like latency, freshness, quality and completeness of the results. In this work, we focus primarily on freshness of results while we consider data streams to have *update semantics*. To date, we have found no framework that defines a clean and systematic way of using update semantics for load management in push-based stream processing. This PhD proposes such a framework, called UpStream. In the face of overwhelming data rates, our solution to directly minimize the impact of overload on QoS is a storage-centric approach using update queues. This paper describes the UpStream framework with respect to its mission, architectural considerations, integration with a state-of-the-art stream processor, progress to date and research directions.

1. INTRODUCTION

Processing dynamic data in real time has been a challenge for many applications including financial services, security monitoring, location tracking systems, sensor data monitoring and so forth. While traditional processing infrastructures, like Database Management Systems (DBMS) are having difficulties keeping up with fast-changing data generated by push-based data streams, several technologies have emerged to respond to the requirements of modern real-time applications. These include Stream Processing Engines (SPE) like Aurora [3] and Borealis [2], STREAM [5] or TelegraphCQ [9]. In an SPE, data flows through continuous query execution plans most commonly involving sliding window operations, filtering, aggregations, unions or correlations ([5]). We have observed that usually, data is aggregated and processed based on certain items of interest from the observed world (e.g. aggregations over sensor readings, where the sensor id is part of the answer). We refer to these items of interest as *keys* and to the nature of queries that preserve key information as *key-sensitive queries*. The ultimate goal of an SPE is to deliver query results in near real-time while keeping up to data rates. As a result, load management is a central focus in this line of research.

In stream processing, load management efficiency has been measured in terms of Quality of Service (QoS) which may depend on the response time (latency), completeness of results (tuple drops) or relevance of the delivered values ([3]). Traditionally, stream processing engines have assumed applications to have append semantics, i.e., results must be delivered based on all (or as much of) the input values and with the lowest possible latency. There are, however, a common set of applications that exhibit *update semantics*. Such applications require the freshest possible results for all the items of interest (i.e., keys). Let us consider an application that monitors the location and movement of cars within a certain region. What is of utmost importance to the car monitoring application is that its queries are run on the latest reportings from all the cars, while getting query answers that reflect all the reportings from each car during a time interval is less of a focus. An application with update semantics relaxes the completeness requirement while maximizing the importance of fresh results. Therefore, there is no need for all input data to be processed. To sum up, at any point in time, a query result which is produced by using the *most up-to-date* input values (per key) is correct and more valuable to the application compared to query results produced by using *all* of the available input values ever existed so far. This is a radically different semantics than the traditional append semantics.

SPEs commonly model streams as unbounded in-memory FIFO queues. In the context of append semantics, queues tend to grow when the system cannot keep up with input data rates, ultimately leading to QoS degradation. To solve this, various load shedding techniques have been proposed which target latency. However, these techniques result in inaccurate query answers and the focus has therefore been on minimizing the degree of this accuracy loss (e.g., number of dropped tuples). For applications with update semantics, such an issue is not important as long as the results are fresh (i.e., not stale). In this respect, *update streams* (i.e., data streams with update semantics) naturally lend themselves to load shedding and we believe that judging the efficiency of load management for update streams is best done in terms of staleness rather than latency and accuracy loss.

In this research, we plan to address the problem of minimizing staleness for streaming applications with update semantics by pushing down the update semantics in the stream processing pipeline. Our solution is a storage-centric approach that is aimed both at directly minimizing QoS degradation when faced with overload and minimizing memory consumption. We propose the UpStream

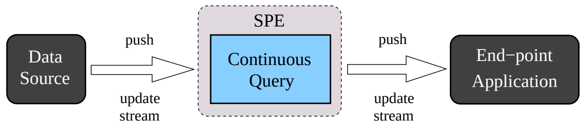


Figure 1: Continuous Query Models

framework, a systematic way of capturing update semantics in a stream processor. As a first step, we have extended the Borealis stream processing engine with *update queues* to incorporate update semantics at the storage level. This offered us an appropriate infrastructure to investigate load management techniques that exploit the characteristics of update streams (i.e. key update patterns and probabilities) or the behavior of the end-point application (i.e. access patterns). Further issues to explore are related to scheduling of multiple queries and adaptive load management in the presence of update semantics. We plan to do this by exploiting modern software and hardware platforms to increase the scalability of a stream processing system.

The rest of this paper is structured as follows. Section 2 gives an overview of the UpStream framework and its progress to date. Then, in Section 3 we discuss some interesting challenges that this research will address further. Finally, we give a brief description of our closest related work in Section 4 and we conclude in Section 5.

2. THE UPSTREAM FRAMEWORK

2.1 Update-based Stream Processing Model

Data streams are generated by push-based Data Sources (see Figure 1) and consumed by Continuous Queries inside a Stream Processing Engine (SPE). We consider such a query to be a directed acyclic graph (DAG) of stream-oriented operators ([13]) linked by arcs that have queues to temporarily store data stream items (i.e. tuples). We assume the query produces results for the End-point Application with a non-negligible processing cost and in a sequential manner. Under update semantics, each result is viewed by the application as an update. Therefore, we interpret the results stream as an *update stream*. Given that each result is produced by the query based on a finite set of input stream tuples, we consider each such set an input update. That is, the continuous query acts on a stream of updates to produce another stream of updates. The type of processing that operators apply on streams affects the scope of an input update. For instance, tuple-based operators (e.g., filter, map) consider individual tuples to be updates, while windowing operators (e.g., sliding window aggregate) consider full windows as updates. As part of our stream processing model, we also consider key-sensitive queries that perform processing based on particular items of interest from the input stream (e.g., a group-by attribute for an aggregation query). In this case, the update stream schema has a field called the *update key*. Updates can only take place for the same update key and key-agnostic queries are treated as single-key queries.

If applications with append semantics care about latency, applications with update semantics focus on *staleness*. Latency captures the total time to deliver a result to the application (i.e. queueing + processing) since the corresponding input update arrived ([3]). Staleness, on the other hand, captures the total time to deliver a result since updates *first* became available. Therefore, by using staleness, we can reason at any point in time about whether (i) the application has a result based on the latest update from the Data Source (i.e. fresh output), or (ii) the current result has been superseded by newer input updates (i.e. stale output). Staleness is measured continuously for every update key.

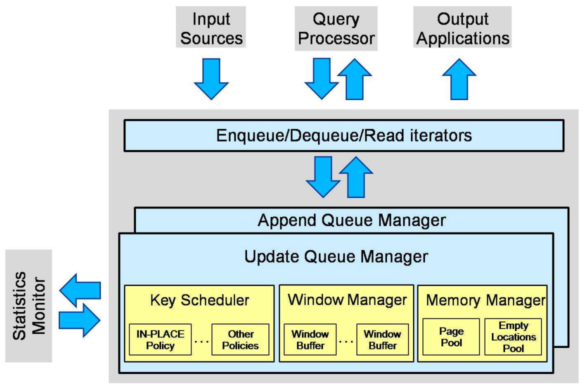


Figure 2: UpStream Storage Manager Architecture

2.2 Storage-centric Architecture

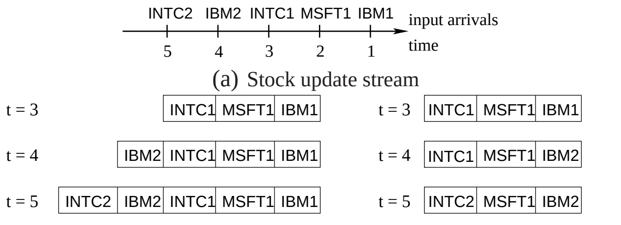
In this work, we take a storage-centric approach to load management for streaming applications with update semantics. Our motivation for doing so is threefold. First of all, storage is the first place that input tuples hit in the system before they get processed by the query processing engine. The earlier the update semantics can be pushed in the processing pipeline, the better it is for taking the right measures for lowering staleness. Second, it is rather easy to capture update semantics as part of a tuple queue. Performing load management at this level prevents the need to store unnecessary data items, which greatly improves memory consumption. 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. This kind of a model is also in agreement with recently proposed streaming architectures that decouple storage from query processing such as the SMS framework [8]. To support the new update-based query processing model, we have extended the traditional append-based model with *update queues*. An update queue groups the input stream by update key (where applicable), and for each distinct update key it is only responsible for keeping the most recent update (i.e., “in-place updates”).

Figure 2 shows the architecture of our storage manager. Storage manager interfaces with input sources, output applications, and Query Processor through its iterators. During runtime, statistics data is produced or retrieved to/from the Statistics Monitor. The Update Queue Manager itself is broken into three main components. In our design, these three components are layered on top of each other and handle three orthogonal issues The Key Scheduler (KS) decides when to schedule different update keys for processing and can employ various different policies for this purpose. Scheduling points are decided by the Query Processor between two consecutive query runs. 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. Basically, the role of the Memory Manager is to allow in-place updates per key group and prevent memory proliferation at overload.

2.3 Key Scheduling

2.3.1 IN-PLACE Update Queue

An IN-PLACE update queue is one that stores the most recent updates for all unprocessed keys and services them in FIFO enqueue order. In Figure 3 we show the behavior of the IN-PLACE update queue (Figure 3(c)) vis-a-vis the traditional append queue (Figure 3(b)), given a stream of tuple-based stock updates, like the



(b) Traditional Append queue (c) IN-PLACE Update queue

Figure 3: Append Queue vs. IN-PLACE Update Queue

one in Figure 3(a). For instance, at time t=4, IBM2 overwrites IBM1 in its original place in the update queue, while in an append queue it would have required a new location for IBM2 at the end. Assuming there is a scheduling point after t=5, we can clearly see that IN-PLACE update queue delivers the most recent update for key IBM and that memory usage was minimal. More generally, the benefits of the IN-PLACE update queue reflect in both load shedding and key scheduling:

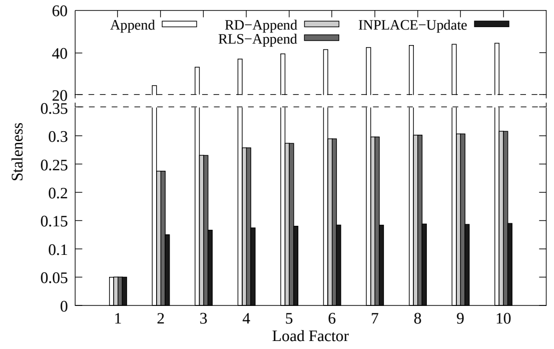
- **Efficient load shedding**: As the default behavior of our update queues, IN-PLACE scales very well with increasing load. The load factor LF is given by the ratio between input rate and processing rate (e.g. LF=10 when results get processed every 100msec and inputs arrive every 10msec). We have experimentally verified this claim in a number of settings, varying load factor and number of keys, using our UpStream prototype. Results for the single key tuple-based case are depicted in Figure 4(a) which shows a comparison between the IN-PLACE update queue and the traditional append queue plus two variants of load shedding based on random drops performed by a specialized operator (RD-Append) and in-queue (RLS-Append), respectively. The update queue tends to stabilize staleness when load increases while the append-based counterparts seem to fall behind. We have also looked at memory usage and observed a drastic improvement as well on the update queue side. When varying the number of keys (Figure 4(b)), both update queue and append-based load shedding counterparts exhibit linear scale-up. In the update queue case, staleness grows because the average queue length, which directly influences the waiting time, is proportional to the number of keys. However, the increase in staleness is much slower for the update queue and this comes primarily from the fact that it keeps only the most recent updates for all keys, which is not guaranteed by the random drop techniques.
- **Efficient key scheduling**: IN-PLACE policy results in the lowest average staleness that can be achieved when the update frequencies of the update keys are uniformly distributed. In brief, this is due to the fact that IN-PLACE policy always chooses the key with the maximum waiting time W, which directly influences staleness. A detailed proof can be found in our technical report ([14]).

2.3.2 Linecutting

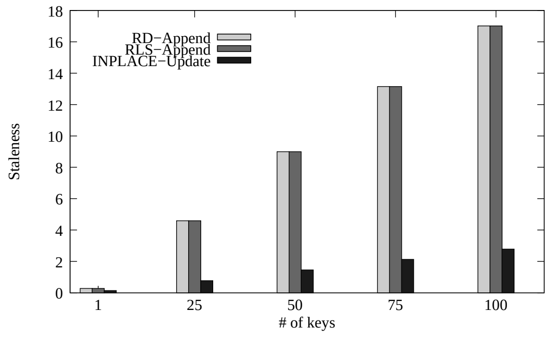
The previous section indicated that IN-PLACE key scheduling policy is the best if all keys update at the same frequency. However, often times the update frequencies are not uniform. We have analyzed such a situation for financial market data taken from NYSE Trade & Quote (TAQ) database for a trading day in January 2006 [1]. More than 3000 different stock symbols (i.e., update keys) were involved, and we observed 49% of them to update rather rarely while the rest of 51% updated quite frequently. Based on these observations, we have raised the following questions: Is IN-PLACE policy still the best that we can do for such non-uniform update frequencies? If not, how can we exploit the differences in update frequencies to find a better scheduling algorithm? To find the answers, we considered a simplified distribution of update probabilities containing two classes of keys: *slow* updaters and *fast* updaters. We assumed all keys contained in a class to update with the same probability while the difference between classes is denoted by what we call the *skew* parameter (e.g. fast keys update 10 times more frequently than slow keys).

In order to build a policy that improves on IN-PLACE we have considered the following observation. IN-PLACE always chooses the key with the greatest waiting time W (now - first enqueue time) regardless of how often that key updates. This causes even slow updating keys to wait the same amount of time on average as the regular ones. To make the key scheduler more aware of update frequencies, we introduced the LINECUTTING heuristic. Basically, LINECUTTING tries to promote slow updating keys to the front of the queue of keys. However, this is not done invariably, so a few rules have to be in place: (1) LINECUTTING should be able to identify the slow updaters with respect to the current state of the queue and (2) Promoting keys to the front of the queue should not starve the keys over which it jumps.

To minimize staleness, LINECUTTING orders the queue of keys based on a sum of factors coming from the above-mentioned design principles, respectively. The first is what we call *slowness* of a key and the second is the waiting time of that key given the current state of the queue. The greater the sum, the closer the key is to being served. Slowness is computed based on the key update rate and its position in the queue. Slowness values are big for slow keys and small for fast keys. Based on this, LINECUTTING has the following effects: (i) the waiting time for slow updaters is reduced, which means their overall average staleness is reduced as well; (ii) fast updaters are still served as in the IN-PLACE case, but with some penalties that come from promoting other keys. That is, we get benefits from slow updaters with penalties on fast updaters. However, LINECUTTING makes sure the benefits are greater than



(a) Single update keys



(b) Multiple update keys (Load Factor 4)

Figure 4: Update Queue vs. Append Queue

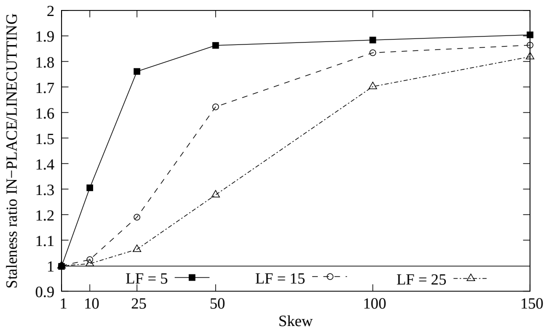


Figure 5: LINECUTTING vs. IN-PLACE

the penalties.

We have experimentally compared LINECUTTING to IN-PLACE in a number of scenarios characterized by various load levels and different two-classes distribution instances (given by number of slow updaters and skew). For instance in Figure 5, we have varied the skew of a symmetric distribution of 20 keys (10 slow, 10 fast) and we observed the improvement offered by LINECUTTING over IN-PLACE (horizontal line at 1 shows zero improvement) for three load levels (LF): 5,15 and 25. The graph reveals a general trend of increasing improvement with skew while increasing the load affects improvement negatively. The former point is explained by slow keys updating less and less as skew grows. However, with increasing load, the update rate of the slow updaters decreases more slowly, which would account for the latter point. Another thing to note here is that the non-improvement at skew=1 (uniform distribution) proves LINECUTTING to be an adaptive heuristic by not making unnecessary promotions. Even though not shown here, we have also experimented with asymmetric distributions (varying number of slow updaters) and we have again seen LINECUTTING doing better than IN-PLACE. However, the amplitude of the improvement depended on the ratio between the number of slow and fast keys affecting the balance between benefits and penalties.

2.4 Windowing Operations

Sliding window queries take as input a window that consists of a finite collection of consecutive tuples and apply an aggregate processing on them, resulting in a single output tuple for that window. In this case, there is no $1-1$ mapping between the input and the output, but an $n-1$ mapping, where n is the number of tuples in the input window. Due to this difference in operational unit, our problem has an additional challenge compared to the tuple-based scenario: Besides worrying about the recency of our outputs, we must also make sure that 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., [20]). In UpStream, an input update is a full window. We adopted the following two design principles for handling windowed updates correctly: (1) Windows should either be processed in full, or not at all (*All or nothing*); (2) If we decide to open a window and start processing it, we must finish it all the way to the end (*No undo*). Changing decisions on a partially processed window is not allowed. In this case, we say that we "committed" to that window.

In [20], load shedding is achieved through a special Window Drop operator which injects window keep/ drop decisions into the input tuples by marking them with window specification bits. These marks are then interpreted by the downstream aggregate operators, which can be arranged in arbitrary compositions (pipeline, fan-out, or their combinations). As a result, subset results are produced. In UpStream, we have essentially pushed the above tuple marking logic down to the storage level, by making our update queue "window-aware". Instead of a window drop operator, the UpStream update queue manager marks the tuples inside the storage before the query processing on those tuples actually starts. As such, our window-aware update queues have one additional advantage over the window drop approach: windows which are redundant (if any) can be identified and their tuples can be immediately pruned inside the update queue before they hit the query processor (this is analogous to the in-place updates in the tuple-based case). Another difference from the window drop approach, consists in the manner of marking which windows to keep and which to drop. UpStream keeps the most recent windows. This is done with the help of window buffers which maintain the data structures for managing windowed updates. There exists one window buffer for each key in the update queue. The Window Manager takes care of window buffers and dictates when to commit to a window: at window starts or window ends. We have come up with two window management policies based on the moment when the commit decision is taken.

- **Lazy Window Buffer**: The first 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.
- **Eager Window Buffer**: The second 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 (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).

Both approaches are designed to deliver fresh results and are superior to append-based approaches. However, from our experiments in the prototype system, we have observed the lazy approach to be more suitable for non-incremental window processing while eager was more suitable for incremental window processing.

3. RESEARCH DIRECTIONS

The previous section introduced the update-based stream processing model and a suitable architecture for an update semantics aware stream processor, UpStream. Our solution is based on update queues which minimize staleness as soon as possible in the processing pipeline and allow for minimal memory usage. Two major features of update queues were presented: key scheduling and windowing operations on update streams. The latter has to do with the effect that update semantics has on operator semantics (like the sliding window aggregate) and vice-versa. While having found a good way to apply update semantics to windows, we are yet to have the same for the rest of the typical streaming operators (e.g. join). We plan to address this as part of future work. Regarding the former feature and not only, we would like to discuss some interesting directions to follow. First, we aim to build a complete scheduling framework. That is, we envision a set of heuristics that match static application requirements to system and environment constraints. Second, after having the scheduling framework in place, we plan to study hybrid strategies that combine various specific scheduling policies to best match changing application requirements. To this extent, we have two research directions to explore in the context of UpStream: *real-time scheduling* and *adaptive load management*.

3.1 Real-time Scheduling

Key scheduling to minimize staleness is a case of real-time scheduling. In this section we want to discuss the research directions regarding key scheduling (per query) and query scheduling (multiple queries) under update semantics using a uni-processor model.

3.1.1 Update Patterns

In Section 2.3 we showed the default behavior of the update queue with IN-PLACE scheduling policy and an improvement in the form of the LINECUTTING heuristic that was targeted at synthetic key update distributions with fast and slow updaters. In this specific scenario, the results are promising and motivate us to investigate further into more generic distributions. For instance, a good immediate step would be synthetic distributions having m classes of update keys. Synthetic distributions drive the search for the good scheduling heuristic. Nevertheless, our heuristics have to be tested against well known statistical distributions, such as Poisson or Zipf.

During our investigation of a good LINECUTTING heuristic, we observed that some configurations of factors (load, number of keys, number of slow updaters, skew) were either very hard to optimize or left very little room to do so. This led us to believe that there has to be a point beyond which no optimizations could be made for a uni-processor model. We know that whatever heuristics we may find, they should behave at least as good as IN-PLACE. Therefore, if IN-PLACE offers the upper bound on performance, we ask what is the lower bound? Unfortunately, such a lower bound can be found only off-line and this may represent a hard task to solve. However, we plan to search for the lower bound in hope to find the optimization space across system configurations (load + key distribution parameters).

3.1.2 Access Patterns

So far, we have assumed that applications are interested in all the keys equally. An example of such applications are the monitoring type. Nevertheless, there are applications that need to monitor only a subset of the keys which are more relevant or more important during a particular time interval. The relevance that the application assigns per key, translates into how often the application looks at the results for that key. For instance, it may happen that sensors in region A have started to produce high volumes of readings all of a sudden. As these sensors represent an increased interest to the application, treating all the keys fairly by the system is not an option. Therefore, the system needs to explicitly prioritize them. Dealing with this new dimension to the problem means we have to revise our scheduling framework and introduce the so-called access frequencies (i.e. priorities) into the model. The result is a system that handles efficiently both uniform and non-uniform update frequencies for a set of relevant keys while relaxing the QoS model for others less relevant.

3.1.3 Multi-query Scheduling

We have analyzed the key scheduling problem concerning queries with one update stream containing multiple updating keys. However, our techniques are not bound to single queries, especially since a stream processor may run multiple user queries at the same time. We believe that key scheduling can be extended to multiple query scheduling and that for each query the storage-centric approach can still bring benefits as shown so far. However, our approach has to incorporate the heterogeneity of queries (e.g. different processing costs, different semantics, etc.) and the different contention patterns in a shared-resources environment. It is our intention to extend the UpStream framework to coordinate multiple update streams with the goal of minimizing staleness of the query answers, in whatever configuration or form queries may be found. As a result, we expect to define a set of efficient *multi-query scheduling* strategies.

3.2 Adaptive Load Management

Adaptive load management refers to the ability of the system to sense changes in the characteristics of the streams, workload or application requirements and employ the correct measures or strategies that respond best to the new setting. In fact, the scheduling heuristics will become the building blocks for adaptive load management techniques. We envision two directions in this respect.

3.2.1 Hybrid Semantics

As a further study in UpStream, we would like to investigate the possibility of employing a hybrid model between append and update semantics. Such a model would process all arrivals for a set of keys while the others would be updating. The update queue is adaptive enough to respond to varying load levels. However, hybrid semantics refers to the ability of the system to switch between update and append semantics based on application requirements. For instance, the application may want to switch focus on a set of keys for which it wants *all* values processed, while caring less about other keys, which are left to update as usual. Hybrid semantics means we may have append-based and update-based processing (i) employed at the same time for distinct key or query sets or (ii) for the same key or query. We expect to come up with a correct configuration of this model and account for the trade-off in staleness for switching back and forth between append and update behavior. Another dimension of the hybrid model would be to react to changes in the characteristics of the streams and application requirements by searching the best key scheduling strategy to match the new configuration.

3.2.2 Elastic processing model

So far, we have discussed UpStream in the context of a uni-processor model with shared system resources. We have shown that UpStream offers guarantees in terms of the recency of processed updates and stable staleness levels with increasing load. However, this may not be sufficient when the application introduces some firm constraints on staleness levels. For instance, the application may impose that results be delivered with staleness that should not go above a certain threshold. However, the system may not always be able to meet this constraint under a uni-processor model. Therefore, while still keeping the update behavior, the system may try to parallelize execution in order to lower staleness below the threshold. This introduces the need for an elastic processing model that would achieve dynamic and minimal resource allocation.

4. RELATED WORK

This research is closely related to work in stream load management. Existing data stream management systems, like Aurora ([3]), treat streams as append-only sequences. Load management in such systems is focused 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 involve both coming up with a good initial operator placement (e.g., [22]) as well as dynamically changing this placement as data arrival rates change (e.g., [6], [23], [15]). In general, moving load is a heavy-weight operation whose cost can only be amortized for sufficiently long duration bursts in load. For short-term bursts leading to temporary overload, load shedding is proposed. In load shedding, the distribution of operators onto the processing nodes is kept fixed, but other load reduction methods (e.g., drop operators, data summaries) are applied on the query plans which results in approximate answers (e.g., [19], [4], [16], [21], [18]). All of these techniques focused on reducing latency for applications with append semantics, and none of them provided storage-based solutions. In UpStream, we take a storage-centric approach to load management. Basically, the system is faced with a resource allocation problem when running key-sensitive queries: when overload occurs, the question is what (keys) should we process first to minimize overall QoS degradation.

A major focus in this research is exploiting application update semantics by embedding them into the streaming infrastructure. Update semantics have also been studied recently in non-streaming domains like synchronizing databases, materialized view maintenance, loading data warehouses and so forth. For instance, Cho and Garcia-Molina study the problem of update synchronization of local copies of remote database sources in a web environment [10]. The problem is when and what to synchronize so as to maximize time-averaged freshness of local copies. Although aiming at a similar problem (staleness is the opposite of freshness), we address an entirely different model which is push-based. This makes the input arrival times known and synchronization is done proactively through continuous queries. Sharaf et al propose a scheduling policy for multiple continuous queries in order to maximize the freshness of the output streams disseminated by a web server [17]. 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. A more recent and closely related work to UpStream is the DataDepot Project from AT&T Labs [12], [11], [7]. While having many warehousing features, DataDepot also deals with real-time scheduling of update jobs in a streaming data warehouse, which is of particular interest to us. However, in UpStream we target continuous operations on streams (e.g., sliding window queries) which can be interpreted as a pre-processing step for a real-time data warehouse system such as DataDepot.

5. CONCLUSIONS

In this paper, we presented UpStream, a framework that deals with load management for streaming applications with update semantics. Such applications care more about having the most up-to-date query results while they relax the completeness requirement. We proposed a novel storage-centric framework for load management based on update queues. As part of our initial and on-going work, we have shown:

- A baseline architecture for an update semantics-aware stream processor which was built as an extension of a state-of-the-art streaming engine, Borealis. The goal here was to find the correct framework that is best suitable for applications with update semantics.
- A set of new techniques for update key scheduling that prove scalable with load. IN-PLACE policy handles best uniform update key distributions, while LINECUTTING is designed to cope with a special case of non-uniform update probability distribution.
- Space-efficient and low-staleness techniques for window processing. It is our plan to also study the behavior of other streaming operators (e.g. joins) under update semantics and thus provide full support for a wide range of queries.

UpStream offers two major research directions: real-time scheduling and adaptive load management. Finally, for more information about the UpStream models, initial implementation and performance evaluation, we would like to refer the interested reader to the UpStream technical report ([14]).

6. REFERENCES

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

[2] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

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

[4] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *ICDE* , 2004.

[5] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3), 2001.

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

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

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

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *SIGMOD*, 2003.

[10] J. Cho and H. Garcia-Molina. Synchronizing a Database to Improve Freshness. In *ACM SIGMOD* , 2000.

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

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

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

[14] A. Moga, I. Botan, and N. Tatbul. UpStream: Load Management Techniques for Update Streams. Technical report, ETH Zurich, Switzerland, http://www.systems.ethz.ch/research/projects /upstream/UpStream-TR.pdf, 2009.

[15] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *ICDE* , 2006.

[16] F. Reiss and J. M. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *ICDE* , 2005.

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

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

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

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

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

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

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