UpStream: A Storage-centric Load Management System for Real-time Update Streams

Alexandru Moga
Systems Group, ETH Zurich, Switzerland
amoga@inf.ethz.ch

Nesime Tatbul
Systems Group, ETH Zurich, Switzerland
tatbul@inf.ethz.ch

ABSTRACT

UpStream is a framework for load management over data streams with update semantics. It provides a novel storage manager architecture that can be plugged into data stream processing engines for serving streaming applications that require low-staleness results over real-time continuous queries. We propose to demonstrate the key aspects of the UpStream architecture as well as its performance using two different application scenarios: One that models a continuously updating financial market dashboard, and another one that is based on an intelligent transportation system for monitoring moving vehicles on a road traffic network. The demonstration will illustrate how UpStream can provide low-staleness query results for these applications under highly overloaded situations, by using a number of update scheduling and storage management techniques. This will be done through a number of interactive visual monitoring tools for the application interface as well as for monitoring the run-time operation of the UpStream system itself.

1. INTRODUCTION

Processing high-volume data streams in real time has been a challenge for many applications including financial services, multi-player online games, security monitoring and location tracking systems. Various load management techniques have been proposed to deal with this challenge from dynamic load balancing to adaptive load shedding. Most of these techniques are best-effort in nature and rely heavily on application-specific resource allocation and system optimization techniques based on Quality of Service (QoS) specifications. In UpStream, we focus on one such application property which characterizes a common set of applications, yet has not been sufficiently addressed by previous work thus far: *update semantics*.

For applications with update semantics, each data element (or a window of data elements) in a stream represents an update to a previous one, and therefore, the most recent arrival is all that really matters to the application [9]. For example, a stock broker watching a continuously updating market dashboard is often times interested in the current market value of a particular stock symbol. Similarly, in systems that involve continuous mobile object tracking such as in Intelligent Transportation Systems, there is a need to monitor the current GPS location of each vehicle as well as the latest average vehicle speed or traffic flow for selected road segments or geographical regions [1]. In such applications, the main goal is to provide *the most up-to-date answers* to the application with *the lowest staleness possible*, as opposed to the applications with the traditional append semantics, where providing all answers with the lowest possible latency is the normal modus operandi. Under normal load conditions, the QoS requirements of these two classes of applications coincide, since lowering latency also lowers staleness. However, under overload, previous load management techniques fall short in satisfying the staleness requirements of the update-based applications since they optimize for latency.

In UpStream, we propose a scalable framework for adaptively managing overload for update-based applications in a way to minimize their staleness and memory usage [9]. Our framework is "storage-centric" in that the update semantics is pushed to the storage level, allowing us to immediately apply in-place updates inside the tuple queues as soon as new events hit the system. This has been one of the key and novel design decisions in UpStream that greatly impacts its performance and usability.

In this demonstration proposal, we first present an architectural overview of our UpStream system, followed by a short related work summary. Then we describe the application scenarios that we are building on top of UpStream to showcase its main features.

2. UPSTREAM SYSTEM OVERVIEW

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

As in the case of append streams, continuous queries on update streams can be composed of a number of stream-oriented operators. In this demo, we focus on two representative operator classes. The first class includes the *state-*

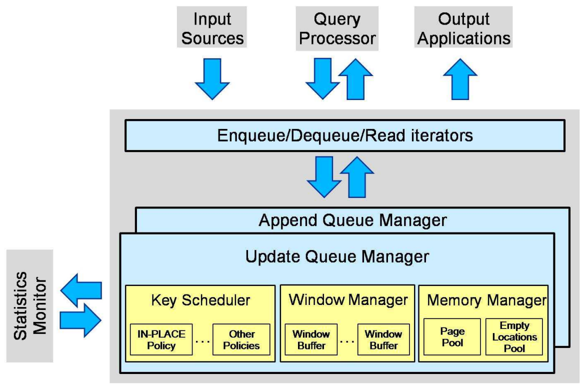


Figure 1: UpStream storage manager architecture.

less operators that execute on a per-tuple basis (e.g., filter, map). The second class includes *stateful* operators that execute on a per-window basis (e.g., sliding window aggregate). The stateful operators map updates on input tuples into updates on windows, each corresponding to an output tuple. This difference in operational units must be taken into account in managing the updates in the system, since output staleness will then apply to whole windows rather than individual input tuples.

One of the key architectural features of UpStream is that it takes a storage-centric approach to handling high-volumes of real-time update streams. At the core of the system is a lossy tuple storage model called an "update queue". An update queue is responsible for keeping only the most recently arrived update for each distinct update key value; older updates can be discarded right away. Furthermore, each key may be updating at a different frequency than others. Similarly, each output application may like to access the query results at different rates. Given these, UpStream must decide which keys should be prioritized in query processing so that overall application staleness can be minimized. This requirement has led to several different flavors of update queues in UpStream (e.g., IN-PLACE, LINE-CUTTING), each suitable for a particular type of workload.

Figure 1 shows a high-level overview of the UpStream storage manager architecture. The Storage Manager interfaces with Input Sources, Output Applications, and the Query Processor through its iterators. These iterators enable three basic queue operations: enqueue, dequeue, and read. Input Sources always enqueue new tuples into a queue, whereas Output Applications always dequeue tuples from a queue. The Query Processor can enqueue intermediate results of operators, while it can read or dequeue these back again to feed into the subsequent operators in the query pipeline. The Storage Manager also communicates with the Statistics Monitor in order to get statistics (e.g., key update frequencies) to drive its optimization decisions. The underlying queue semantics can either be append or update. In this demo, we will 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 (in-place updates, etc.). The Window Manager (WM) takes care of maintaining the window buffers according to the desired sliding window semantics. Finally, the Memory Manager (MM) component oversees the physical page allocation from the memory pool. In our design, these three components are layered on top of each other and handle three orthogonal issues: KS is responsible for minimizing staleness, WM is responsible for correct window processing, and MM is responsible for the efficient management of the available system memory where the actual data is physically stored [10].

We have implemented UpStream on top of the Borealis stream processing engine, expanding on its Storage Manager component [3]. The QoS and statistics monitoring components of Borealis were also extended in order to keep track of update key frequencies and to compute staleness and memory usage. It should be noted that the UpStream design is general enough to be applicable on any stream processing engine. This power comes from the fact that it follows a design that decouples storage management from the query processing engine [5].

3. RELATED WORK SUMMARY

UpStream mainly relates to previous work in three broad research areas: (i) stream load management, (ii) synchronization and freshness in web databases, and (iii) materialized view maintenance. A more detailed summary of related work in these areas is provided in a recent publication [9]. However, we would like to discuss here in detail, one recent work in (iii) which relates to UpStream quite closely: The DataDepot Project from AT&T Labs [8]. DataDepot is a tool for generating data warehouses from streaming data feeds, thus it has many warehousing-related features. For us, the part on real-time update scheduling is directly relevant. We see two basic similarities between UpStream and DataDepot: Both accept push-based data and both worry about staleness. On the other hand, in DataDepot, updates correspond to appending new data to warehouse tables. Therefore, all updates must be applied. Furthermore, DataDepot focuses on scheduling the update jobs, but does not consider continuous operations on streams (e.g., sliding window queries). UpStream could potentially serve as a pre-processor for a real-time data warehouse system such as DataDepot. Thus, the two works are complementary.

4. DEMO SCENARIOS

In this section, we describe two application scenarios that will demonstrate different key aspects in UpStream.

4.1 Financial Market Dashboard

Our first application is from the financial services domain. We took a real financial dataset from NYSE TAQ (Trade and Quote) database [2], which we can replay as a real-time continuous data stream. This dataset contains two types of streams: Trades and Quotes. A trade tuple contains the last price and volume of a stock traded for a particular stock symbol at a particular time. A quote tuple contains the current highest bid and the current lowest ask for a stock symbol. Therefore, both Trades and Quotes streams have update semantics. On these data streams, we can define a number of continuous queries that also exhibit update semantics. For example:

```
Q1: For each symbol, report the current price and the total
    change in price relative to the beginning of the day.
Q2: For each symbol, report the total trade volume and the
    average trading price in the last 10 minutes.
```

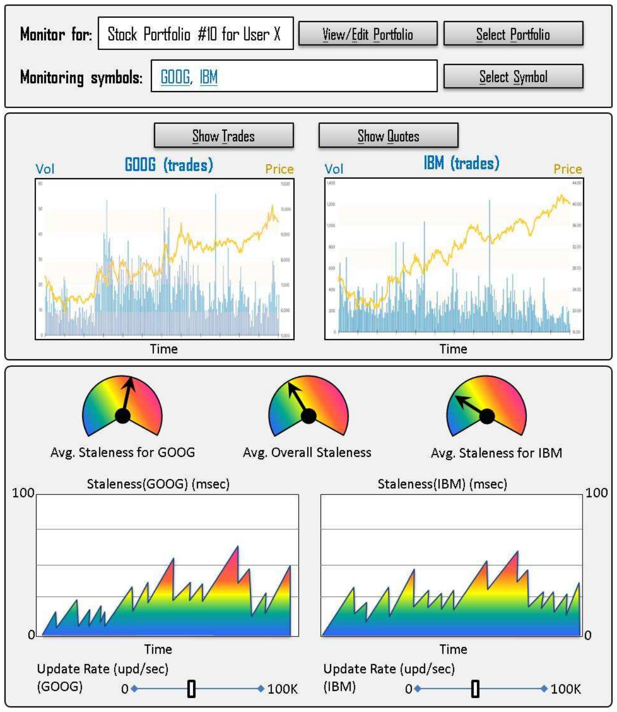


Figure 2: Real-time dashboard for UpStream.

Q1 produces an update stream, where the key is the stock symbol and the scope is the time elapsed since the beginning of the day; whereas in Q2, the window of interest consists of the last 10 minutes from the Trades stream for the same symbol. Such queries need to provide answers that reflect the most recent state of the input. Otherwise, decisions based on stale data may lead to loss in profits.

We will use this application scenario to demonstrate how UpStream can reduce staleness via *update frequency-aware key scheduling*. We analyzed the NYSE TAQ dataset (e.g., for a trading day in January 2006) and saw that update frequencies for different stock symbols over a given time period were not uniform. We exploit differences among update frequencies of keys in making intelligent scheduling decisions in the tuple queues. Thus, in addition to our in-place update queue policy (which we have theoretically and experimentally proven to be the best possible approach for uniform update frequencies [9]), we will also demonstrate line-cutting policies, which essentially allow slowly updating keys to cut in front of the other keys in the update queue. We have shown that this can indeed lead to a great reduction in the overall query staleness over the in-place policy [9].

As continuous queries, we are planning to use sliding window aggregation queries with grouping by symbol (similar to Q1 and Q2 shown above). The results of the continuous queries will be visualized on a real-time dashboard along with their staleness values, as shown in Figure 2. We will have several *stock portfolios* containing a number of stock symbols to be monitored. The stream of symbols from the NYSE TAQ dataset will be fed into UpStream at different rates. For instance, we can have diversified portfolios: Some symbols show intense trading, i.e., high update rate, while others less intense, i.e., low update rates. We can monitor selected symbols from the selected portfolio (top panel in Figure 2) both in terms of query answers (middle panel) and most importantly in terms of staleness levels (bottom panel). For the latter, we can observe staleness levels in real-time and average staleness per symbol and overall when different optimization techniques are employed in UpStream.

4.2 Road Traffic Monitoring

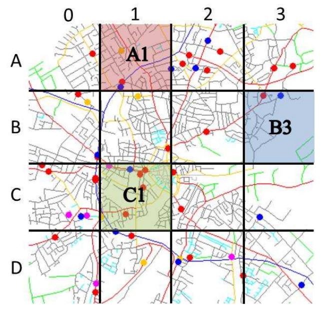
In this scenario, we model a real-time monitoring service for a transportation system, inspired by initiatives such as ITS [1]. Here we specifically focus on monitoring vehicles traveling inside a metropolitan area. However, the type of queries and techniques shown in this scenario are general enough to address continuous monitoring of any kind of moving objects.

Continuous monitoring of moving objects consists of *spatio-temporal queries* (e.g., range, nearest-neighbor, or time-para metrized [7]). A moving object generates position reports of the form *(time, object-id, coordinates, speed, ...)*. A stream consisting of such position reports for all moving objects has clear update semantics: Each new report is an update on the previous one, for the same object-id value. A continuous query on the stream of position reports can also exhibit update semantics. For instance, a typical query may be to retrieve the available cabs that are *currently* within 1 mile of 33 N. Michigan Ave., Chicago [11]. On the other hand, such a query needs to provide results in a timely fashion even when the update rates are very high. As also pointed out by Wolfson et al [11], high update rates can be a reason for inherent uncertainty in moving object databases. In UpStream, this uncertainty can be easily captured by *staleness* of the query results.

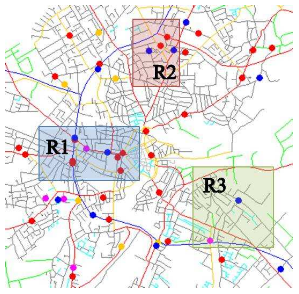
In this demonstration scenario, we plan to show how UpStream can reduce the staleness of spatio-temporal queries under conditions of high load. UpStream can do so via efficient *update-aware load shedding* directly on the input stream. In recent work [9], we have introduced the window-aware update queues, which can perform update-aware load shedding for sliding-window aggregation queries. We also showed through experimental analysis how the window-aware update queues reduce staleness in a better way than state-of-the-art approaches, i.e., random-based window-aware load shedding. Here we use them for spatial queries that produce results using a time-based sliding window, such as the following:

```
Q3: Continuously report the number of cars that have passed
    through region R in the last hour.
Q4: Continuously report the number of cars in region R.
```

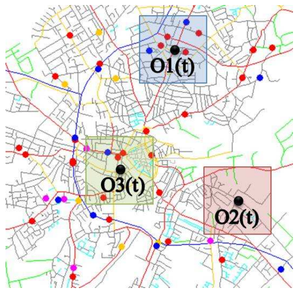
From a temporal perspective, we consider Q3 to be a *historical* query since it reports based on what has happened in the last period of time. On the other hand, we call Q4 a *snapshot* query since it reports based on the current positions of the moving objects. From a spatial perspective, we envision the following queries on spatial data: *predefined*, *user-defined*, and *mobile range* queries on one hand, and *k-nearest neighbor* (kNN) or *skyline* queries on the other. Figure 3 shows several screenshots taken from our visualization tool that allows us to depict the results of spatio-temporal queries running in UpStream as well as controlling query parameters. The figures show only a part of the GUI that is in charge with displaying the entire space (in this case the map of a city), the location of moving objects and the regions surrounding the query points.



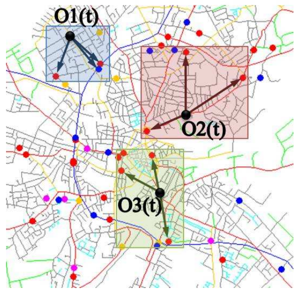
(a) Predetermined ranges.



(b) Arbitrary fixed ranges.



(c) Mobile ranges.



(d) kNN and skyline.

Figure 3: Spatio-temporal queries in UpStream.

▶ **Predefined Range Queries.** Ranges are defined based on information that can be found in the input stream. In other words, we can assume that the entire space is divided into non-overlapping regions (i.e., sectors), as depicted in Figure 3(a). In this case, the input stream has an additional field, the *sector-id*, which indicates the sector where a moving object has reported position (e.g., A1, B3, C1 in Figure 3(a)). The query can compute a number of aggregations (e.g., object counts, center of mass, average speed, ingress/egress counts etc.) based on the objects found in each sector. This way, the output stream is an update stream where the update key is the sector.

▶ **User-defined Range Queries.** This time, ranges are defined by the user. They are arbitrary and fixed (e.g., regions R1, R2, and R3 in Figure 3(b)). Normally, each range would entail a separate query that can perform the same type of aggregations as in the case of predefined range queries. However, UpStream benefits from Borealis' group-by sliding-window aggregate operator. Therefore, we can combine all range queries into one which produces an update stream where the update key is the range itself.

▶ **Mobile Range Queries.** We define a mobile range query to be object-centric. Therefore, the range moves with the object. For instance, Figure 3(c) shows the ranges around the positions of the objects O1, O2, and O3 at time *t*. The query performs aggregations based on the objects within range, and produces an update stream where the update key is the object itself.

▶ **k-Nearest Neighbor and Skyline Queries.** These queries are similar in style to the mobile range queries. They are object centric, i.e., the output is an update stream where the update key is the object. However, instead of a user-defined range around the object, the query scope is made up by the k-nearest neighbors or the skyline (i.e., the most interesting set of objects [4]) in relation to the object. An example query would be to maintain the minimum bounding box around each object and its k-nearest neighbors. Figure 3(d) depicts this for objects O1, O2, and O3 at time *t*.

The demonstration will be carried out based on the visualization tool which shows the query points and query scopes as well as the moving objects themselves (to some extent) directly on the map of a city in real-time. One can use the tool to change the query parameters or select specific points (e.g., ranges or objects) of interest to be displayed. Staleness information from UpStream can also be shown through variable transparency of the regions around the query points (e.g., if region R2 in Figure 3(b) is completely transparent, then the latest aggregation result for that region has zero staleness). The data streams will be generated using a network-based generator of moving objects such as the one described in [6]. Different load levels will be exerted onto the system by varying the input stream rates.

5. ACKNOWLEDGEMENTS

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.

6. REFERENCES

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

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

[3] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR Conference*, pages 277–289, Asilomar, CA, January 2005.

[4] S. Borzsony, D. Kossmann, and K. Stocker. The Skyline Operator. In *IEEE ICDE Conference*, pages 421–430, Heidelberg, Germany, April 2001.

[5] I. Botan, G. Alonso, P. M. Fischer, D. Kossmann, and N.Tatbul. Flexible and Scalable Storage Management for Data-intensive Stream Processing. In *EDBT Conference*, pages 934–945, Saint Petersburg, Russia, March 2009.

[6] T. Brinkhoff. A Framework for Generating Network-based Moving Objects. *GeoInformatica*, 6(2):153–180, June 2002.

[7] J. Dittrich, L. Blunschi, and M. Vaz Salles. Indexing Moving Objects using Short-lived Throwaway Indexes. In *SSTD Symposium*, pages 189–207, Aalborg, Denmark, July 2009.

[8] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream Warehousing with DataDepot. In *ACM SIGMOD Conference*, pages 847–854, Providence, RI, June 2009.

[9] A. Moga, I. Botan, and N. Tatbul. UpStream: Storage-centric Load Management for Streaming Applications with Update Semantics. *VLDB Journal*. to appear.

[10] A. Moga, I. Botan, and N. Tatbul. UpStream: Storage-centric Load Management for Data Streams with Update Semantics. Technical Report TR-620, ETH Zurich, March 2009. ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/620.pdf.

[11] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *IEEE SSDBM Conference*, pages 111–122, Capri, Italy, July 1998.