Continuous Analytics: Rethinking Query Processing in a Network-Effect World

Michael J. Franklin, Sailesh Krishnamurthy, Neil Conway, Alan Li, Alex Russakovsky, Neil Thombre
Truviso, Inc.
1065 E. Hillsdale Blvd, Suite #230, Foster City, CA 94404
www.truviso.com

ABSTRACT
Modern data analysis applications driven by the Network Effect are pushing traditional database and data warehousing technologies beyond their limits due to their massively increasing data volumes and demands for low latency. To address this problem, we advocate an integrated query processing approach that runs SQL continuously and incrementally over data *before* that data is stored in the database. Continuous Analytics technology is seamlessly integrated into a full-function database system, creating a powerful and flexible system that can run SQL over tables, streams, and combinations of the two. A continuous analytics system can run many orders of magnitude more efficiently than traditional store-first-query-later technologies. In this paper, we describe the Continuous Analytics approach and outline some of the key technical arguments behind it.

1. INTRODUCTION
Modern network- and web-based applications are pushing traditional database and data warehousing technologies beyond their limits [2]. At the heart of the problem are two complementary workload characteristics that have combined to challenge the accepted store-first-query-later approach employed by traditional and even alternative database architectures such as column stores and data warehouse appliances: massive data growth and increasing demand for lower latency.

1.1 Network Effect #1: More Data
Companies across all industries are seeing very steep increases in the amount of data they must process. For example, one recent study [13] has estimated that the amount of data stored in data warehouses has been growing by an average of 173% per year across all industries. This rate of growth is substantially faster than the typical 12 to 18-month doubling of hardware capacity as dictated by Moore's law, Shuggart's law and others. As a result, for data analytics workloads hardware continues to become slower relative to the demands being placed on it.

As severe as this problem is in traditional businesses, however, the problem is even more acute for companies in network-centric businesses such as social networks, advertising networks, content delivery, e-commerce, on-line gaming, and security. Many companies in these industries are facing (or at least, planning for) data volume growth of as much as 10x per year. In such environments, "peak" load one year quickly becomes "normal" load the next, and this process continues. These increases are driven by viral network-effects that lead to hyper-growth of user bases and by the competition-driven need to add new features coupled with application development advancements that enable the rapid deployment of such features.

With existing data analysis approaches, sustaining even a couple years of massive compounded growth, if even possible, would require an investment in hardware, management, and electrical power, (or the equivalent in payments to cloud resource providers) that would be far beyond the means of all but the very largest of enterprises.

1.2 Network Effect #2: Less Time
Exacerbating the data growth problem is a continual downward pressure on latency for analytics. Network-centric businesses must react quickly to changes in their environments and workloads and to the demands of their users. For on-line businesses, understanding what a user is doing while they are still interacting with the site provides the opportunity to improve user experience as well as to more accurately target advertising and offers. Furthermore, across many industries, sophisticated data analytics are increasingly a core source of competitive advantage.

Surprisingly, despite their interactive nature, most "modern" web-based companies face analytics latencies similar to those of older industries – even though they are not burdened by legacy IT infrastructure. In most cases, next-day reporting and analysis is still considered to be state-of-the-art. This state of affairs frustrates business managers at these companies, while the IT managers fret over how to maintain even such a loose latency requirement in the face of massive data growth.

1.3 Problem: A Decades-Old Legacy
For many environments, it has become increasingly apparent that the data warehouse is a bottleneck in the analytics pipeline (see for example, [11]). It is our belief that this problem is not simply a matter of tweaking existing data warehousing products. Rather, it is an inherent by-product of the traditional *store-first-query-later* nature of data management and database architecture. That is, batch-oriented processing, in which data is first collected, then cleaned, then distributed and/or stored, then retrieved, then analyzed, is just fundamentally too inefficient to handle the analytics challenges faced by modern network-centric businesses.

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/3.0/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2009.
4th *Biennial Conference on Innovative Data Systems Research (CIDR)* January 4-7, 2009, Asilomar, California, USA.

While traditional DBMS architecture is obviously challenged in such environments we note that even new "disruptive" approaches like Hadoop and Map/Reduce are also based on a batch paradigm. Thus, they too suffer from inherent inefficiencies that render them exceedingly costly and slow for many common analytics tasks.

1.4 Workload Characteristics

Fortunately, modern analytics workloads have key attributes that can be exploited to solve the dual Network Effect problems of too much data and not enough time.

First of all, these analytics applications tend to be "additive" in nature. That is, rather than consisting of transactional updates to an existing set of data, these applications absorb constantly arriving streams of data, with new analytics applied primarily to the newly arriving data. For example, an on-line business may wish to monitor site usage, referral or buying behavior, content interaction, etc. as people use the site. As in most analytics workloads, the time attribute plays a central role in the analysis. Metrics of interest are computed on the current data over different time-scales and reported as is or perhaps compared to the same metrics over time periods in the past.

Secondly, as is typical of analytics and reporting workloads, in these applications the vast majority of queries and metrics of interest are known ahead of time. That is, the Key Performance Indicators that are needed by the business users are typically well known for a given organization, and new metrics are added only occasionally. While as in any analytics environment, a small number of power users will perform *ad hoc* data mining types of analysis, the majority of the analytics processing is done on the core metrics. When data mining detects a new phenomenon of interest, that insight typically identifies new metrics to be monitored from then on. Furthermore, such *ad hoc* analysis can often be done much more efficiently on previously computed metrics rather than on the raw data that has been archived away in the database or file system.

We argue that these dual characteristics of additive, time-oriented data and known queries provide an opportunity for solving the widening scalability gap for database technology. We propose the seamless integration of stream-oriented Continuous Analytics into the data management platform as the solution. We call such an integrated solution a "stream-relational" database system.

2. STREAM-RELATIONAL SYSTEMS

In this section we describe the basic idea of a Continuous Analytics system based on Stream-Relational principles.

2.1 Optimizing for the Common Case

A standard principle of systems design is to *optimize for the common case*. That is, a system should be designed to be most efficient for the situations that are expected to arise the majority of the time. Unfortunately, the approach underlying modern Relational query processing systems is optimized for a case that is certainly no longer the common one for analytics systems today, if in fact, it ever was. Namely, the existing store-first-query-later approach is aimed for situations in which a large database that is randomly updated by transactions is manipulated by a query workload that is at best unpredictable in its timing, if not perhaps, purely *ad hoc*.

In contrast, as discussed in the previous section, the common case for analytics in network-centric scenarios has neither of these attributes. That is, the workloads are additive (i.e., append-mostly) and the queries and their scheduling are largely known in advance.

Stream Query Processing has been designed for workloads of *additive*, time-oriented processing where queries and metrics of interest are known ahead of time. As such, it provides a perfect technology for addressing the crisis in analytics being caused by the Network Effect. That is, by embedding a high-performance stream query processor into a full SQL-based relational DBMS, a unified *stream-relational* system can be created that is capable of processing even batch-oriented analytics and reporting workloads many orders of magnitude faster than alternative approaches. This degree of scalability improvement is what is required to address the dual problems of massive data volume growth and demands for low latency of network-centric applications.

Continuous Analytics, then, can be viewed as a rethinking of Relational DBMS query processing for this new "common case". If one were to develop a system optimized for running predefined, time-oriented (or sequence-oriented) analytics over continuously arriving streams of records, a store-first-query-later approach does not make sense. Likewise, a pure streaming approach in which data is passed through the system once and answers are spewed out in real-time [7] is not adequate for a reporting-oriented analytics workload, as the on-line use of historical data is crucial to such applications.

2.2 Why an Integrated Solution?

The argument for using stream query processing to address the Network Effect problem is straightforward. Stream processing systems execute queries incrementally over data "on-the-fly". This approach leads to huge efficiency benefits by processing queries over data without first having to store that data, and by processing multiple continuous queries in a shared manner. We refer to this continuous, incremental processing as "Jellybean Processing": Rather than first filling up the jellybean jar only to later pick through the jar to calculate metrics about the contents (e.g., how many beans there are of a particular flavor), it is hugely more efficient to simply calculate all those metrics simultaneously as the beans are being put in the jar. Such processing avoids the costs of reading and writing to/from disk, moving data repeatedly through the memory and cache hierarchy, starting up and tearing down query state, etc. and enables redundant work to be avoided across the set of active queries [4,12].

The argument for not using just a standalone stream query system is also straightforward. Stream query systems are typically designed solely for real-time applications involving monitoring, alerting, and/or event detection. These systems often have query languages that beyond superficial similarities differ sharply in semantics and capabilities from standard SQL. Their use typically requires business processes that are structured specifically for monitoring and utilizing real-time data. Unfortunately, most existing business applications are not yet able

to cope with real-time data, and for many applications, real-time data simply is inappropriate. Existing analytics and reporting applications have been built assuming a standard database interface in which SQL queries are submitted to the system and answers are delivered according to the reporting policies and schedule of the given use case.

2.3 Realizing an Integrated Approach

At the heart of the Continuous Analytics approach is a seamless merger of streaming and traditional Relational query processing. From a language point of view, this merger is surprisingly easy to do. The key is to begin with the philosophy of making minimal extensions to the existing SQL approach while not losing any of its well-understood functionality or behaviors. With thought, even concepts such as transactional consistency can be extended to continuous processing. The result of this approach is a system that is inherently familiar to experienced database developers and DBAs and that fits nicely into existing applications while providing dramatic performance and scalability improvements.

From a technology point of view, the merger is more involved. The key concept here is that streaming data and stored data are not intrinsically different. Rather, stored data is simply streaming data that has been entered into persistent structures such as tables and indexes. Starting from this core principle, it becomes possible to construct a system that processes queries over streams and tables in a way that does not require database users to overhaul their thinking or database administrators to learn a completely new set of concepts in order to leverage the performance and scalability benefits of this new technology.

And as a side benefit – a Continuous Analytics system can provide “real-time” processing for those applications that are equipped to take advantage of it.

3. DESIGN CONSIDERATIONS

In this section,, we provide a brief overview of our Stream-Relational extension of SQL and outline some of the implementation benefits that result from this approach.

3.1 Query Langauge Overview

Much of the early research on stream processing systems was ambiguous about the relationship between stream and relational query processing. In fact, some early systems did not support a query language at all [1] or introduced imperative syntax such as “for loops” [5]. In contrast, the STREAM project [2] introduced the Continuous Query Language (CQL), which explicitly defined this relationship. CQL was based on formal relational languages and used the explicit relationship between relations and streams primarily as a way to provide clear and consistent semantics for streaming queries. In our language, called TruSQL, we have taken this approach further, by actually integrating stream processing fully into SQL, including persistence. In other words, TruSQL is a superset of SQL.

We add the notion of *streams* to the standard relational model. A stream is an ordered unbounded relation. For instance, the following DDL example shows the definition of `url_stream`, a stream that is ordered on an attribute called `atime` where each record represents a URL and the IP-address of the client machine that accessed that URL at that time.

```
CREATE STREAM url_stream
( url         varchar(1024),
  atime       timestamp  CQTIME USER,
  client_ip   varchar(50),
);
```

Example 1 - DDL For Creating a Stream

In TruSQL, queries can be posed exclusively on relations, exclusively on streams, or on a combination of streams and relations. In the first case, a query produces a relation as an output and has the exact same semantics as in SQL. We refer to queries over relations as *snapshot queries* (SQ) because they operate on a snapshot of the relations at a given time. In the other cases, however, a query produces a stream as an output. Since a stream is unbounded, a query that produces a stream never ends and is therefore called a *continuous query* (CQ). SQ’s produce an answer and terminate (i.e., they are regular relational queries) while CQ’s produce answers incrementally and run until they are explicitly terminated.

Because streams are unbounded, when a stream is used in a SQL query, the system must be told how and when to consider the data in the stream. This is done using a *window clause*. The window clause effectively creates a sequence of relations from the stream, and the SQL query is applied to each of these relations. This is known as “RSTREAM” semantics in CQL [2].

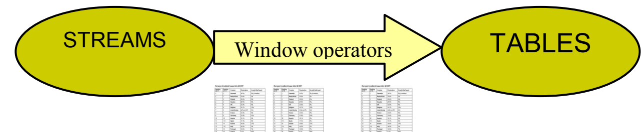

Figure 1 – Windows Produce a Sequence of Tables

In TruSQL there are several different types of window clauses, but a typical window clause specifies the width of the window and how often it is moved forward. These can be specified using time, row counts or combinations of the two. For example, the query below is a CQ over the url_stream that each minute (specified by “ADVANCE”) produces the top ten urls visited over the previous five minutes (specified by “VISIBLE”). Note that the only extension to the SQL syntax here is the window clause for the stream.

```
SELECT url, count(*) url_count
FROM url_stream <VISIBLE '5 minutes'
                 ADVANCE '1 minute'>
GROUP by url
ORDER by url_count desc
LIMIT    10
```

Example 2 - A Simple Continuous Query (CQ)

The result of a stream-only or mixed (i.e., streams joined with relations) query in TruSQL is a stream formed by concatenating the sequence of relations produced by repeated execution of the query. Of course, as in a traditional database system, data independence allows for efficient implementations of these semantics under the covers.

3.2 Composition: Derived streams and Views

An important benefit of building on SQL is the ability to leverage the query composition features of the language. TruSQL supports two approaches for composing CQs: Streaming Views and Derived Streams. A Streaming View is simply a SQL view defined using a CREATE VIEW statement with a query that includes one or more streams. A Derived Stream is a stream object defined by a CREATE STREAM AS statement.

As with relational views, a query that defines a Streaming View is only instantiated when the view is itself used in another query. In contrast, a query that defines a Derived Stream runs in an "always on" mode until it is explicitly dropped. For example, consider the DDL statement below:

```
CREATE STREAM urls_now as
SELECT url, count(*) as scnt, cq_close(*)
FROM url_stream <VISIBLE '5 minutes'
                 ADVANCE '1 minute'>
GROUP by url;
```

Example 3 – Creating a Derived Stream

This statement creates a derived stream called `urls_now` that continuously computes a streaming query. It returns, each minute, the number of appearances of each URL in the `url_stream` over the previous five minutes. [1]

Note that the results produced by `urls_now` are always available within at most one minute. A derived stream is, therefore, particularly useful for clients that operate in a disconnected fashion since the results of a CQ are available upon the first window close after a client re-connects to the system. Also note, however, that the `urls_now` stream, as defined in this example is not archived. That is, as specified, its results are simply discarded if there is no active CQ using the stream. We show how to create an archived Derived Stream next.

3.3 Active Tables: Persistence Meets Streams

Another important advantage of the approach of incorporating SQL is that it becomes simple to provide persistence to streams using standard SQL tables. Such tables are truly full-fledged SQL tables, and can be used in the same ways that any SQL table can be used. For example, consider the following DDL statements:

```
CREATE TABLE urls_archive
(url varchar(1024),
 scnt integer,
 stime timestamp);

CREATE CHANNEL urls_channel
FROM urls_now
INTO urls_archive APPEND;
```

Example 4 - Persistence for Streams: Tables and Channels

The first statement is a standard SQL CREATE TABLE statement that creates a table called `urls_archive`. The second statement is an extended DDL statement that creates a special Channel object called `urls_channel` that is responsible for storing the Derived Stream `urls_now` into the `urls_archive` table. Note that in this case, "append" semantics are being used so that new results are simply added to the table. Another option is "replace", in which each new result from the stream overwrites the previous result. Since `urls_archive` is kept continuously updated, we call it an Active Table.

The table `urls_archive` is a SQL table that can be used as any other table in a SQL query. For example, it can be queried as part of a report generation process. The advantage over the traditional approach of running the report on the raw data only after it has been stored, of course, is that the reporting query will run extremely fast, as the computation has already been done. And because Active Tables are simply SQL tables, indexes can be defined over them to further improve query performance. Thus, the combination of Derived Streams with Active Tables can be viewed as an extremely efficient materialized view mechanism; One that leverages modern, shared stream query processing to explicitly address the requirements of additive analytics in Network Effect environments.

Active Tables are also the key for enabling continuous queries that compare current metrics with past metrics. Such a query is written simply as a join between the stream and the active table:

```
select c.scnt, h.scnt, c.stime
from (select sum(cnt) as scnt,
      cq_close(*) as stime
      from urls_now <slices 1 windows>) c,
      urls_archive h
where c.stime –'1 week'::interval = h.stime
```

Example 5 - Stream-Table Join for Historical Comparisons

The above discussion covers only a small portion of the TruSQL language, but it demonstrates the ease with which stream query processing and traditional relational query processing concepts can be combined at the language level and identifies some of the important benefits of doing so. While other stream processing approaches have embraced SQL to varying degrees, we believe that beyond simply adopting some of the SQL syntax, there are tremendous advantages to be had by carefully integrating stream

[1] The "cq_close(*)" function returns the timestamp at the close of the relevant window.

and traditional SQL functionality. This seamless Stream-Relational integration is the key to the Continuous Analytics approach.

4. IMPLEMENTATION ADVANTAGES

A unified stream-relational language that minimally extends SQL provides obvious benefits to database administrators and application developers who are already proficient at using SQL database systems to solve their analytics and reporting problems. The minimalist approach, however, has important benefits in terms of systems implementation as well. These benefits stem from the fact that existing concepts and techniques for query processing, transactional semantics, high-availability and such can be extended to work in the continuous analytics setting. Perhaps of equal importance, however, is that by unifying streaming data with relational data it becomes possible to leverage large portions of existing DBMS code to build such a system, thereby avoiding the reinvention and redevelopment of functionality that often takes a decade or more to get right. For example, the CQ query plans in many cases are able to reuse the existing implementations of standard, well understood, iterator-style relational query operators (e.g., filters, joins, aggregates, sort).

Another important area of reuse is in the transactional and recovery subsystems. Since a CQ essentially runs as a long-running transaction, and can involve tables as well as streams, a major semantic issue that needs to be addressed is the visibility rules with respect to updates of tables. The isolation mechanisms of some RDBMSs, such as multi-version concurrency control can be extended to provide continuous isolation semantics that are meaningful in a streaming environment. For example, a notion of *window consistency* that ensures that updates to tables are visible only on window boundaries [6].

Likewise, recovery is a key problem that needs to be solved in any system that is intended for use in a mission-critical fashion. Unlike a traditional RDBMSs, that only guarantees the integrity of durable state (all in-flight transactions are deemed aborted on failure), a Stream-Relational system needs to recover runtime state as well as durable state. In a single-node implementation, this runtime state must be rebuilt from data persisted on-disk. While a common approach for this kind of state recovery is to periodically checkpoint the internal state of the various CQ operators, such an approach is hard to implement correctly and requires every operator to be "taught" how to recover its state. Using the concept of Active Tables, it is possible to instead implement a strategy that rebuilds runtime state from disk automatically. Such an opportunity is yet another example of the implementation benefits to be gained by carefully following an approach of integrated Stream-Relational processing.

Continuous Analytics can be used anywhere existing SQL-based processing is used for reporting and analytics. As such, the range of use cases spans the breadth of existing data warehouse and business intelligence applications as well as new applications where such techniques have been deemed inappropriate due to their high cost and high latency.

The benefits of the approach can be huge. For example, in one scenario for a network security reporting application, a batch-oriented query taking over 20 minutes using a database system (which was one of a suite of dozens of queries that needed to be run several times a day), was produced in milliseconds (yes, this is a 5 orders of magnitude speed up!) by simply running the query continuously and incrementally as the data arrived, and storing the results in an Active Table for later retrieval. The effort involved in converting this static query to a "jellybean" query was measured in minutes, and the overall architecture of the solution remained unchanged – a standard database was simply replaced by a SQL-compliant Stream-Relational database system.

5. RELATED WORK

Continuous Analytics is a technique for addressing the analytics and reporting demands of modern network-centric enterprises. It is an extension of traditional database approaches to analytics, and as such, there is much related work both in terms of techniques that can be incorporated and in terms of alternative approaches.

Continuous Analytics clearly derives from the intense activity in stream processing that has been an important thrust of the database research community in recent years. In particular, academic prototypes such as Aurora, Nile, Stream, and TelegraphCQ [1, 5, 3, 9] examined many aspects of stream query processing, and developed key concepts for efficient query processing. The focus of these research efforts was largely on "real-time" applications such as large-scale sensor networks, financial trading, and telecom. Numerous companies and open source efforts have followed the initial research, and by and large, these efforts have focused on the same set of problems.

A key difference between this earlier work and our approach, however, is that the earlier systems all treated stream processing as distinct from traditional relational (i.e., persistent) query processing. We believe that this narrow scope has been a key inhibitor to the more mainstream adoption of stream processing techniques in many areas. Furthermore, by separating the stream world from the table world, huge opportunities to leverage existing skill sets, tool sets, IT infrastructure, and code are lost. The Stream-Relational approach we advocate here is aimed at reclaiming these benefits while providing the necessary performance for the massive data analytics problem being faced by network-centric enterprises.

Another important related technology is that of materialized views (MVs) [14]. MVs were developed precisely to address the inherent inefficiencies of store-first-query-later database technology for query-heavy, non-*ad hoc* workloads observed in many analytics applications. MVs, however, are not optimized for such workloads. They still require storing the data, are not optimized for high data arrival rates, they do not exploit shared processing for the full SQL language, and they do not fully exploit the time-oriented semantics of the data and queries in modern analytics workloads.

This latter limitation is fundamental. MVs are refreshed in batch mode and therefore may be out of date at the time of the query. There are limited means to tell the system when to update; mostly on a timer or upon transaction completion. And when the update

starts, the whole batch is processed. Even if the DBMS is clever enough to process the changes incrementally, disk operations, trigger mechanisms and transaction management take significant time even before processing has started.

By contrast, the Continuous Analytics approach calls for processing the data as it arrives utilizing the available CPU cycles, so by the end of the appropriate time window the answer is ready. Streaming windows offer more control over update scheduling. Since the system knows that the query is evaluated continuously, there is no need to reissue the query. But the most important difference is that stream processing takes advantage of the fact that incoming data is ordered without the need to create and maintain on-disk indices.

In some sense, Continuous Analytics can be viewed as a next generation MV mechanism exploiting the full power of stream query processing technology and the full power of a traditional RDBMS to address the scalability demands of Network Effect applications.

Finally, many organizations are using or experimenting with data parallel processing technology such as map/reduce or Hadoop [8]. Such technologies are effective at bringing massive amounts of processing to bear on data crunching problems, but are inherently batch-oriented and are much more resource intensive than the Jellybean processing that a stream-relational system can provide. They also have low-level interfaces that can make application development and maintenance difficult.

Recent systems such as Hive [9] are addressing the interface language issue by putting SQL interfaces on top of such libraries. Such projects, as well as the increasing number of SQL-based parallel database and data warehousing products raise the possibility for closer integration between Continuous Analytics systems and more batch-oriented approaches. A key to such integration, however is how faithfully each of the systems conforms to the SQL interface.

6. CONCLUSIONS

Modern network- and web-based applications driven by the Network Effect are pushing traditional database and data warehousing technologies beyond their limits due to their massively increasing data volumes and demands for low latency. To address this problem, we advocate an integrated query processing approach that runs SQL continuously and incrementally over data *before* that data is stored in the database. This Continuous Analytics technology is seamlessly integrated into a full-function database system, creating a powerful and flexible system that can run SQL over tables, streams, and combinations of the two.

A continuous analytics system can run many orders of magnitude more efficiently than traditional store-first-query-later technologies. Because of their Stream-Relational nature, such systems can efficiently support a range of workloads from batch-reporting to "real-time" monitoring. They can also support workloads that need to combine streaming and table-based data, both for enriching fact data with table-based dimension data and for comparing current metrics with historical ones.

Perhaps most importantly, because they do not break the existing interface models for data analytics, Continuous Analytics systems built using a Stream-Relational approach can be easily integrated into existing IT environments while providing the extreme scalability and performance demanded by modern data-intensive applications. As such, the approach represents a long-needed rethinking of query processing in light of the new "common case" for data analytics.

REFERENCES

[1] Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S. Aurora: A New Model and Architecture for Data Stream Management. In *VLDB J* (12)2: 120-139, August 2003.

[2] Agrawal, R., et al. "The Claremont Report on Database Research", http://db.cs.berkeley.edu/claremont/, May 2008.

[3] Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, (15)2, June 2006.

[4] Arasu, A., Widom, J. Resource Sharing in Continuous Sliding-Window Aggregates. *Proceedings of VLDB 2004.*

[5] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin M., Hellerstein, J., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *Proceedings of CIDR 2003*.

[6] Conway, N. Transactions and Data Stream Processing. http://neilconway.org/docs/thesis.pdf, April 2008.

[7] Garofalakis, M., Gerhke, J., Rastogi, R., "Tutorial: Querying and Mining Data Streams, You only get one look", *Proc. ACM SIGMOD 2002.*

[8] Hadoop Web Page, http://hadoop.apache.org/core/, 2008

[9] Hive Web page, http://hadoop.apache.org/hive/, 2008

[10] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh , M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, X. Xiong, "Nile: A Query Processing Engine for Data Streams", *Proc. ICDE Conf.,* 2004.

[11] Kobielus, J, ""Really Urgent Analytics: The Sweet Spot for Real-Time Data Warehousing," Forrester Research Report, August, 2008.

[12] Krishnamurthy, S., Wu, C., Franklin, M. On-the-fly sharing for streamed aggregation. *Proceedings of SIGMOD 2006.*

[13] Winter, R, "Why Are Data Warehouses Growing So Fast?", *B-eye Network*, http://www.b-eye-network.com/view/7188, April 2008.

[14] Widom, J., ed., Special issue on Materialized Views and Data Warehousing, *Data Eng. Bull., June 1995.*