TruSQL: A Stream-Relational Extension to SQL

Submitted for Conference Review May 6, 2008

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

ABSTRACT

Stream query processing systems perform queries, analytics, and monitoring over data arriving at unprecedented rates. To date, however, most stream processing systems have either eschewed SQL, or have been based on "SQL-like" languages, thereby forfeiting many of the technology and business benefits derived from SQL's years of use as an industry standard. Truviso takes a different approach, namely we start with a complete SQL implementation and extend it to incorporate high-performance, on-the-fly stream processing. This approach leads to a very different kind of stream processing system, one that is fully integrated with traditional persistent data processing, allowing applications to span streaming and historical data in a seamless manner. We refer to this as a Stream-Relational DataBase Management System (SRDBMS). In this paper we describe the extensions we have made to SQL that enable SRDBMS functionality. We believe that these extensions can serve as a useful model for industry efforts towards seamlessly adding stream processing to standard SQL.

1. INTRODUCTION

Stream query processing has emerged as one of the most disruptive data management technology advances in many years. Stream processing systems execute queries incrementally over data "on-the-fly", providing instantaneous analysis, filtering, monitoring and alerting over streams arriving at very high rates. More importantly, stream processing 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. These performance advantages enable stream technology to address the data overload crisis being faced by organizations across the spectrum of data-intensive applications.

Surprisingly, despite this potential for huge impact, most stream processing efforts to date have been focused only on applications where data arrival rates are off-the-charts and near real-time response is required. While stream processing is indeed uniquely able to address such applications, this focus has obscured the much larger opportunity for stream processing, namely, that it represents a re-invention of query processing and analytics for *all* data-intensive environments.

Truviso's approach to stream processing is uniquely aimed at addressing this larger opportunity. Specifically, Truviso has built a "Stream-Relational" database system (SRDBMS) that extends standard SQL to provide a high-performance platform for data-intensive applications.

1. SQL as "Intergalactic Dataspeak"

Our approach is motivated by the experience of previous "disruptive" innovations in database technology. For example, the stream processing revolution echoes the Object-Oriented Database (OODB) revolution of the late

1980's[4]. OODB technology was developed in response to the limitations of traditional relational database systems for addressing emerging (at that time) applications. The work started in academia and research labs and quickly spawned a number of innovative start-ups. Driven by the focus on these emerging applications, OODB vendors designed systems with new languages that somewhat resembled, but differed substantially from SQL and furthermore, built new systems that were not able to provide the basic levels of functionality and performance that customers of RDBMS platforms had come to expect. An alternative approach, called Object-Relational database systems (ORDBMS) incorporated the key ideas from the OODB efforts, but did so in a way that did not break the relational paradigm, and allowed for a smooth migration for existing applications that were based on SQL. As stated at the time "For better or worse, SQL is Intergalactic Dataspeak"[11]. This pronouncement has largely been borne out in the database industry since then. More recently, a similar story has been playing out with XML-oriented database systems.

The SQL language has proven to be tremendously resilient and adaptable over the years, incorporating extensions for Objects, XML, On-line Analytical Processing (OLAP) and many others. Truviso has taken this approach to the next step, by incorporating streams and stream processing into SQL to build a *Stream-Relational* DataBase Management System (SRDBMS). That is, rather than developing a query language that merely looks-like SQL or that was inspired by SQL, we started with a full SQL implementation (in our case, SQL as implemented by the PostgreSQL database system[10]) and added stream processing to it. This approach provides substantial benefits, including:

• Existing SQL applications can be run natively.
• SQL programmers do not have to "unlearn" SQL idioms and features.
• Key features of SQL such as data independence, query composition, subqueries, views, transactional semantics, indexes, and optimization are preserved.
• Queries and applications that span streams and tables can be written seamlessly.
• Interaction with legacy relational database systems can be achieved easily.

Furthermore, from a practical perspective, building on an existing RDBMS code base circumvents the 10-year "baking" process required to build an industrial-strength data management system, thereby avoiding many of the pitfalls of earlier database revolutions.

2. A Unified Stream-Relational Approach

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 TruSQL, we have taken this approach further, by actually integrating stream processing fully into an existing relational query language, namely SQL. In our Stream-Relational (SR) language, called TruSQL, streams are added to SQL as a first-class concept. Thus, in TruSQL, one can pose queries directly over streams, tables, and combinations of streams and tables. A TruSQL query containing no streams is simply a (traditional) SQL query

In addition to our main tenet of creating a unified languageas a superset of SQL, we have made several other key design decisions that simplify many of the conceptual problems of previous stream query languages and extend the available opportunities for query optimization. These include:

- We generalize the notion of order in streaming queries to include time and row count, and allow streams to be accessed using either or both of these.
- Windows over streams in TruSQL allow the specification of both width and movement, independently.
- We develop isolation semantics for visibility of updates for queries that combine streams and tables (so called "mixed" queries).

- The default "stream to relation" semantics in TruSQL is "RSTREAM" [2]. In terms of semantics, each application of a window over a stream in a query produces a relation, over which the query is executed. Thus, the result of a continuous or mixed 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.

In the remainder of this paper we describe the TruSQL extension to SQL with a focus on the above aspects.

2. DATA MODEL

In this section we propose the *stream-relational* (SR) extension to the well-known relational model and describe how it is realized in TruSQL. First, we present a model for streams. Next, we describe how to realize streams using a Data Definition Language (DDL) that is analogous to the standard SQL approach for creating tables. Finally, we show how to populate streams using the standard SQL Data Manipulation Language (DML).

1. Streams and relations

The relational model for data management is organized around the notion of a *relation*, which is generally defined as a finite set of tuples that varies over time.[1] A query in the relational model operates on one or more relations and produces another relation as output. Since such queries operate on a snapshot of the database at a fixed point in time, we call them *snapshot queries* (SQ).

In contrast, the stream-relational model adds the notion of *streams* to the standard relational model. A stream is an ordered unbounded relation. For instance, the following example shows the definition of `quotes`, a stream that is ordered on an attribute called `qtime` where each record represents a `price` and `volume` quote for the instrument described by the `symbol` attribute.

```
CREATE STREAM quotes (
            symbol varchar(10),
price             float,
volume integer,
qtime timestamp ORDERED
);
```

In the stream-relational model, 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. In the latter 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). The semantics of CQs are described in detail in Section 3.

All CQs rely on an implicit or explicit ordering of the data in the streams they operate upon. An explicit ordering is expressed by declaring that one or more of the attributes of a stream are "ordered" (e.g., the `qtime` attribute above); this is analogous to a constraint declaration in standard SQL. In addition to any explicit ordering, every stream also has an implicit *sequential* ordering that is based on the arrival order of records in the stream.

The following example TruSQL query computes the average price of all instruments over 100 rows at a time of the `quotes` stream. This particular CQ relies on the stream's implicit sequential ordering.

```
SELECT   symbol, avg(price)
FROM     quotes <VISIBLE 100 ROWS
                ADVANCE 100 ROWS>
GROUP BY symbol;
```

In practice, since the tuples in a stream often represent events that occurred in the real world, the creation time of an event is a common ordering attribute. As a result, we generally refer to an ordering attribute of a stream as its *time* attribute. While the time of a stream is typically based on an atomic data type such as an integer or

a timestamp, it can have other more complex representations, e.g., composite attributes and non-contiguous data types, which are described in Section 6.

2. Data Definition Language

We now describe how TruSQL extends the Data Definition Language (DDL) to allow the creation of streams.
In a traditional RDBMS structured data is organized as sets of tuples in named objects that are called *tables*. A table is therefore a basic building block of the system and represents a relation that varies over time as the table is manipulated by insert, update, and delete operations. Relations can also be formed "on-the-fly" using different kinds of objects, e.g., a *view* that is defined as an SQ on other relations, or a *table function* that is defined using a user-defined function that returns a finite set of rows based on its input arguments. Likewise, TruSQL supports both *raw streams,* that are typically populated by client applications, and *derived streams* and *views* that can be populated by other CQs. In this section, we limit our focus to raw streams and address the other variants in Section 4.
TruSQL incorporates the traditional SQL DDL statements for creating database objects. This DDL is extended with a `CREATE STREAM` command that allows new streams to be defined. In the previous example, a stream named `quotes` is created with 4 attributes, one of which (`qtime`) is designated as the stream's ordering attribute using the `ORDERED` clause. It is often desirable for the system to assign a timestamp to incoming data rows. For this scenario, the SQL standard "generated columns" feature can be used by including the phrase `GENERATED AS clock_timestamp()` [2] before the `ORDERED` clause.
While the notion of a stream as a distinct first class object helps in understanding the semantics of CQs, there is in principle no reason why a stream cannot be formed from a traditional table, or a table cannot be formed from a stream. We describe these transformations later in this paper and show how they unify the notions of streams and relations. In fact, the unification of streams and relations is a key feature of TruSQL that we believe has been missing from prior proposals for stream query languages.

3. Data Manipulation Language

We now describe how to populate and manipulate a stream in TruSQL. As with the DDL, we begin with the SQL Data Manipulation Language (DML) that is used to modify tables in standard SQL.
A traditional RDBMS supports INSERT, UPDATE, and DELETE commands on a table. In addition, most systems also support a mechanism for bulk loading data into tables (in PostgreSQL this operation is called COPY). With TruSQL a stream can be populated using INSERT and COPY statements. The SRDBMS must merely ensure that the ordering constraint of the stream is maintained by requiring that every record inserted or copied into the stream has an ordered attribute that is not smaller than its predecessor.
The following example shows how records can be inserted into the `quotes` stream:

```
INSERT INTO quotes VALUES
('EUR/USD', 1.34, 4000, '10:00 am'),
('EUR/USD', 1.45, 4500, '10:00 am'),
('EUR/USD', 1.32, 5000, '10:10 am');
```

However, the semantics of UPDATE and DELETE operations are more complicated. For instance, what does it mean to update or delete a record that was inserted into the stream in the past? Clearly it is too late to change any results of CQs that have already been consumed by an application. Also, what does it mean to insert a record with an "old" ordered attribute in violation of the stream's ordering constraint? While there are several alternatives for addressing these issues, we do not cover them further in this paper.
Having described the SR data model and TruSQL's implementation of it, we now turn to the TruSQL query model, which integrates both streams and relations.

3. QUERY MODEL

In this section we describe TruSQL query model, which inherits and builds upon many of the ideas pioneered in CQL. The design of TruSQL was driven by two major goals:

1. Streams and relations should be first-class citizens and co-equal peers in the data and query models. This is a direct result of the system we set out to build: one that implements the SR model and not an engine aimed solely at event stream processing.
2. The query language for CQs should be SQL and not a proprietary language or a "SQL-like" language. In other words, the language should consist of coherent extensions to standard SQL that might feasibly be included into the ANSI and ISO standards. As a result, the query language should be easy to learn for anyone who is already familiar with SQL.

In this section, we begin by outlining the execution model for queries in TruSQL. Next, we describe the essential characteristics of windows and propose several different window operators. After reviewing some example queries, we then discuss partitioning, a technique that creates "virtual streams" based on the value of one or more stream attributes. Finally, we describe several attributes that are defined by windows and are often useful for expressing continuous queries.

1. Continuous query execution model

We now describe the philosophy that drives the execution model of CQs. For ease of exposition, in this section we will only consider CQs that involve a single select-project-join-aggregate (SPJA) query block. We consider composition of continuous queries in Section 4.

The basic syntax of a continuous query is identical to a query in SQL, except that the FROM clause can reference both streams and relations.[3] Each CQ contains three components:

1. *Stream-to-Relation (StoR):* This operator describes a deterministic and consistent way to transform a stream into an unbounded sequence of relations. We colloquially describe StoR operators as "window operators", and call each relation produced by a StoR a "window."
2. *Relation-to-Relation (RtoR):* This (composite) operator transforms one or more relations into another relation. An RtoR operator is simply an SQ.
3. *Relation-to-Stream (RtoS):* This operator transforms an unbounded sequence of relations into a stream. The simplest RtoS operator is RSTREAM, which concatenates successive relations together to produce a query's output stream. RSTREAM is the default RtoS operator in TruSQL, and is the only RtoS operator we discuss in this paper.

Syntactically, in TruSQL a StoR operator is enclosed in angle brackets and is placed after the FROM clause entry for the stream to which it is applied. The syntax of a CQ is demonstrated by the following example, which produces the 15-second average price for each stock, every 3 seconds:

```
SELECT   symbol, avg(price), avg(volume)
FROM     quotes <VISIBLE '15 seconds'
                 ADVANCE '3 seconds'>
GROUP BY symbol;
```

The output stream produced by a CQ that operates on an input stream S and a set of stored relations T1,…,Ti can be computed using the following simple algorithm:

1. Apply the StoR operator associated with S to produce an unbounded sequence of relations W1,W2,…,
2. For each relation Wi in {W1,W2,…} apply the RtoR operator to the set of relations {Wi,T1,…Tk} to form an unbounded sequence of output relations O1,O2,…
3. Apply the RtoS operator to the sequence O1,O2,… to produce the query's output stream.

While this model is intuitively clear, the devil is in the details – i.e., in providing a rich set of StoR and RtoS operators that can be used to express a wide variety of queries. We focus on StoR and RtoS constructs in the next two sections – our query language supports the full set of SQL constructs in the RtoR operator.

2. Window operators (StoRs)

Recall that a StoR produces an unbounded sequence of relations from a stream. This suggests that a StoR operator describes two fundamental characteristics: *when* new relations in the sequence are produced, and *what* each relation contains. In this section, we define language elements for describing these window characteristics in a flexible and intuitive way. This flexibility is a key attribute of TruSQL.

A StoR operator specifies a series of contiguous subsequences of a stream. The width of each subsequence is called the *visible interval* of the StoR operator, whereas the condition that must be satisfied for a new relation to be emitted is called the StoR's *advance interval.* For example, this TruSQL query contains a StoR clause that emits a new relation for every 2 input rows; each relation contains the most recent 5 rows in the stream:

```
SELECT * FROM
quotes <VISIBLE 5 ROWS ADVANCE 2 ROWS>;
```

Semantically, the evaluation of a StoR operator is straightforward:

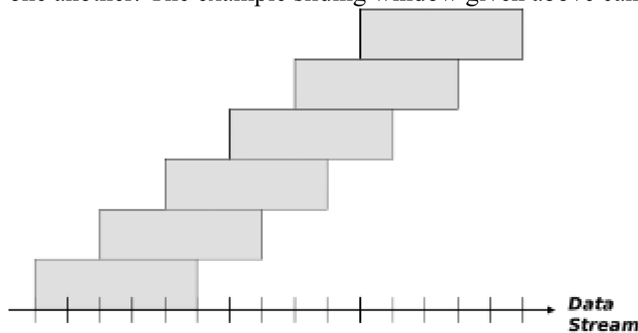
- Input rows are read from the StoR's associated stream
- When the StoR's advance interval is satisfied, the StoR emits a relation that contains all the tuples within the most recent visible interval in the input stream

In the remainder of this section, we describe three variants of the basic StoR concept. We then describe the units of measure that are used to specify intervals within streams. Finally, we illustrate these concepts with a series of examples.

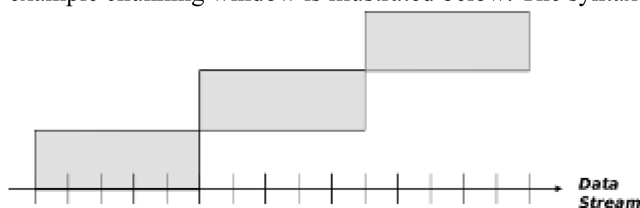
1. *Window types*

We first describe the different kinds of window operators that can be applied to a stream.

Sliding: A sliding StoR has two parameters, an advance interval and a visible interval, that behave as described above. Note that if the visible interval exceeds the advance interval, the relations emitted by this operator overlap one another. The example sliding window given above can be illustrated as follows:

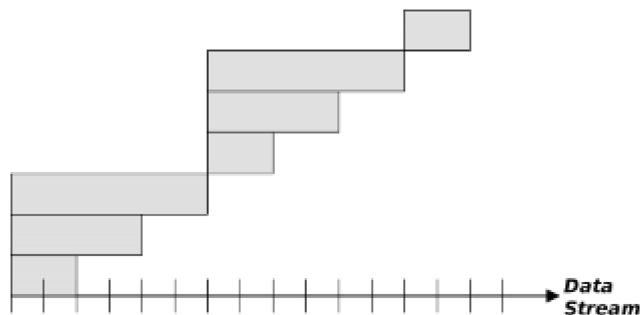


Chunking: This window operator is equivalent to a sliding window whose advance and visible intervals are the same. Therefore, the relations emitted by a chunking StoR are always disjoint. While chunking windows do not offer any additional expressive power, we include them in TruSQL because they arise frequently in practice, are easy to understand, and are amenable to different evaluation techniques than more complex window types. An example chunking window is illustrated below. The syntax for this window is <SLICES 5 ROWS>.



Landmark: This window operator has two parameters: an advance interval and a *reset interval*. Whenever the landmark StoR's reset interval is satisfied, the StoR is said to have been "reset". At each advance interval, the StoR emits *all* the tuples since the last reset occurred. A landmark StoR is considered to be initially "reset"; if no reset interval is specified, the landmark StoR is never reset again. In effect, a landmark StoR is a sliding

window operator whose visible interval is variable and potentially unbounded. An example landmark window is illustrated below. The syntax for this window is `<LANDMARK RESET AFTER 6 ROWS ADVANCE 2 ROWS>`.



2. *Stream Intervals*

StoR operators require a way to describe distances in a stream. We call a unit for measuring such distances a *stream interval*. TruSQL supports 3 types of stream intervals:

1. *Rows*: This interval describes a contiguous subsequence of a stream containing a fixed number of rows. For example, an interval of this type might be used to divide a stream into relations containing exactly *k* rows each. Note that unlike other window interval types, row-based intervals do not require a stream to have an explicit ordered attribute. In effect, a row-based interval operates upon an implicit "row number" associated with each tuple of the stream, where the row number is based on the order in which rows arrive in the stream. This implicit row numbering is further described in Section 3.5.
2. *Time*: This interval describes a contiguous subsequence of a stream, such that the difference in the value of an ordered attribute between the first and last records of the subsequence does not exceed a fixed value. For example, an interval of this type might be used to divide a stream into relations containing 5 seconds of data each. If a stream does not have an explicit ordering attribute, StoRs applied to the stream cannot contain time-based intervals. If a stream has multiple explicit ordering attributes, the StoR clause must specify the ordering attribute to which a time-based interval should be applied.
3. *Distinct Time*: This interval describes a contiguous subsequence of a stream containing a fixed number of distinct values of an ordered attribute of the stream. For example, an interval of this type might be used to divide a stream into relations that each contain *k* distinct values of the ordering attribute.

In TruSQL, these interval types can be freely combined within a given StoR operator – there is no requirement that the advance, visible or reset intervals be specified in the same unit of measure. Unlike systems that offer more constrained window operators, the combination of these different interval types in TruSQL allows windows to be produced from streams in a highly flexible and expressive manner. This allows the simple definition of a wide variety of window flavors, as we demonstrate in the next section.

3. **Examples**

We now present a series of examples that demonstrate the use of TruSQL windows in practice.

Chunking window advancing every 5 seconds. The following example calculates the average price and volume for every symbol that is seen in a 5 second window.

```
SELECT symbol, avg(price), avg(volume)
FROM quotes <VISIBLE '5 seconds'
             ADVANCE '5 seconds'>
GROUP BY symbol;
```

Landmark window advancing 100 rows resetting after 1 hour. The following example calculates the total volume of shares traded after every 100 quotes received. The count is reset every hour.

```
SELECT sum(volume)
FROM quotes <LANDMARK
             RESET AFTER '1 hour'
              ADVANCE 100 rows>;
```

Sliding window advancing every 5 seconds, with 10 seconds visibility. The following example calculates the minimum and maximum price every 5 seconds seen in a 10 second interval for each symbol.

```
SELECT symbol, min(price), max(price)
FROM quotes <VISIBLE '10 seconds'
             ADVANCE '5 seconds'>
GROUP BY symbol;
```

Edges with Different Window intervals. The following example calculates the maximum amount of the last 10 trades calculated every second. This demonstrates a window operator with a time-based visible interval and a row-based advance interval.

```
SELECT max(price*volume)
FROM quotes <VISIBLE 10 ROWS
             ADVANCE '1 second'>;
```

Stream joined with Table. Assuming we have the historical data about minimum and maximum prices for the previous year in a table called `last_year_data`, the following example calculates every 10 seconds how much off prices are as compared to last year's maximum price for each symbol.

```
SELECT q.symbol, (q.price - t.max_price) as delta
FROM quotes AS q <VISIBLE '10 seconds'
                  ADVANCE '10 seconds'>,
    last_year_data AS t
WHERE q.symbol = t.symbol;
```

Stream joined with Stream. Assuming we have another stream called options, the following example tracks every 30 seconds how the maximum price of the symbol GOOG compares with the maximum premium of its January 2009 calls for strike price of $630 in those same 30 seconds.

```
SELECT max(q.price), max(p.premium)
FROM
quotes AS q <VISIBLE '30 seconds'
             ADVANCE '30 seconds'>,
options AS p <VISIBLE '30 seconds'
              ADVANCE '30 seconds'>
WHERE q.symbol = p.symbol
AND   q.symbol = 'GOOG'
AND   p.expire = '2009-01-16'
AND   p.strike_price = 630;
```

4. Partitioned streams

The previous sections described windows that enabled query processing over contiguous subsequences of a stream. Many applications, however, require processing over numerous "virtual" streams. For example, a

network monitoring application may need to monitor traffic between specific pairs of hosts, or a financial trading application may need to calculate metrics over the quote streams of individual stocks. For such applications, *partitioning* can be used to “demultiplex” the virtual streams.

Partitioning splits an input stream into one or more sub-streams (*partitions*) based on a partitioning key. A window operator is then applied to each of these sub-streams independently. Partitioning is enabled by using the `PER` keyword in the StoR clause followed by a “partitioning key” as shown in the example below.

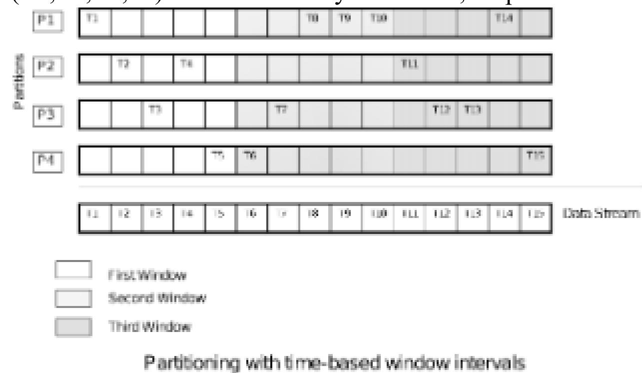
```
SELECT symbol, avg(price)
FROM quotes <PER symbol
             VISIBLE 5 ROWS
             ADVANCE 5 ROWS>
GROUP BY symbol;
```

Note that partitioning takes place before the StoR operator is applied. After the stream has been partitioned, a StoR operator is applied to each of the partitioned sub-streams independently. Each partition emits a new window when the advance interval for *that partition* is satisfied. Partitioning can be used to construct relations from non-contiguous portions of an input stream.

1. *Time-based partitioning*

If the window interval is based on the value of an ordered attribute, then as soon as the window interval is closed by an incoming tuple, then *all* partitions will emit their relations. The rationale for this behavior is that whenever a tuple belonging to a partition closes the window interval for that partition, it can be used to synchronize and close all other partitions since we will never see any tuple with a value that falls in the window interval that we just closed. Therefore, for time-based window intervals all partitions close windows and emit relations in lockstep.

The following diagram provides an example of how windows are formed in partitioning with time-based window intervals of ‘5 seconds’ for a stream where the data tuples are arriving every second in the order shown (T1,T2,T3,…).. Note that every 5 seconds, all partitions P1 through P4 emit a new window.



Partitioning with time-based window intervals

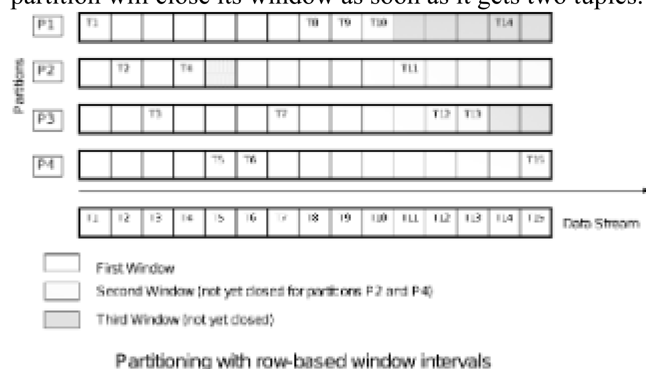
Example: The following query produces the top two quotes for every symbol that arrives in a 5 second window.

```
SELECT symbol, price
FROM quotes <PER symbol
             VISIBLE '5 seconds'
             ADVANCE '5 seconds'>
ORDER BY symbol, price DESC LIMIT 2;
```

2. *Row-based partitioning*

In case of row-based window intervals, however, the behavior is different. A partition can only close its window once it sees a specified number of rows in that partition. Hence, if an incoming tuple closes a window interval for a partition, then only that partition will emit a relation. An input tuple can close a window interval for at most one partition in row-based window intervals.

The following diagram illustrates an example of how windows are formed in partitioning with a row-based window interval of 2 rows for a stream where the data tuples are arriving in the order shown. Note that each partition will close its window as soon as it gets two tuples.



Partitioning with row-based window intervals

Example: The following query produces the average price over the last 5 trades for every symbol.

```
SELECT symbol, avg(price)
FROM quotes <PER symbol
            VISIBLE 5 ROWS
            ADVANCE 5 ROWS>
GROUP BY symbol;
```

3. *Partition expiry*

Depending on the distribution of the partitioning key, it is possible that some partitions might (most probably always will) get less data than others. It can cause the following problems:

- There could be long periods of inactivity in certain partitions, so there has to be a way to purge partitions.
- Recall that in value-based window intervals, all partitions close their window intervals in lockstep. So if no data came into a partition in a window interval it will emit an empty window. This is undesirable.

TruSQL engine provides three different ways to expire partitions:

Explicit Expiry: TruSQL allows the users to specify an expire interval using the `EXPIRE` keyword in the partitioning clause of a StoR. Based on the value specified after the `EXPIRE` keyword, the engine will automatically expire the partitions that did not have any activity in the last expire interval that was closed. This will purge inactive partitions in row-based window intervals and in value-based window intervals of landmark type too. This explicit form of expiry is applicable to both value-based and row-based window intervals.

Conditional Expiry: TruSQL gives the user the flexibility to expire partitions whenever a certain condition is satisfied by the input tuple. The expiry condition does not have to be on window's ordering attribute. Conditional expiry is specified by using the `EXPIRE WHEN` *`condition`* clause while specifying partitioning for StoR. The engine evaluates the condition for every incoming tuple; if the condition is satisfied, it will expire the partition for which the tuple was destined.

Implicit Expiry: For time-based window intervals, the TruCQ engine will expire partitions that may produce empty windows. This is done by the engine without specifying anything explicitly, hence it is called implicit expiry.

5. Row and window properties

The query execution model for CQs involves an SQ that is applied repeatedly to a sequence of windows (relations). Naturally, the SQ has access to all the user-defined attributes for each record in the window. In addition, TruSQL defines several additional "hidden" attributes that are defined for each row in a window. We

say that an attribute is a *row property* if it varies for each row in a window; otherwise, the attribute has the same value for every row in a window and is termed a *window property*. TruSQL defines the following hidden attributes:

1. **cq_rowid**: This row property exposes the implicit sequential ordering that is derived from the order in which tuples arrive at the system.
2. **cq_winid**: This window property exposes the implicit sequential ordering that is derived from the order in which windows are emitted by the StoR operator in the CQ.
3. **cq_close**: This window property exposes the leading edge of each window in the stream.
4. **cq_open**: This window property exposes the trailing edge of each window in the stream.

4. COMPOSITION

In the previous section, we introduced the TruSQL query model, in which a continuous query takes one or more streams as input, and produces a single output stream. This naturally suggests composition as a technique for constructing complex queries: the input to one continuous query can be the output of one or more continuous queries. The ability to compose continuous queries is a powerful feature of TruSQL.

The key difference between composition in relational systems and composition of streams is that relational queries have no temporal extent; they produce a single result set that is invariant over the lifetime of the query. Relational systems take advantage of this property to “pull up” subqueries into joins, push predicates down into subqueries, and lazily evaluate subqueries as warranted. In contrast, continuous queries are unbounded in duration, and produce a continuous stream of results. This provides significantly less scope for optimization. The key difference is that while relational subqueries are dependent objects, streaming subqueries operate independently from their parent query. The parent query merely consumes the output stream produced by the subquery.

In this section, we describe several methods for composing continuous queries, and related features that allow more flexible query composition.

1. Composition mechanisms

Composition of continuous queries is realized with two principal techniques: composition in the DDL with *derived streams* and composition in the DML with *continuous subqueries*.

1. *Derived streams*

A derived stream is a database object with an associated continuous query. As long as the derived stream exists, its associated continuous query is evaluated by the system[4]. The results of that query can be utilized as an input stream by other continuous queries, by referencing the derived stream database object. Note that because the output stream of a derived stream is continuously evaluated by the system, a query that accesses the results of a derived stream can take advantage of state that has been accumulated since before the query itself began execution.

Derived streams are created using `CREATE STREAM AS`. For example:

```
CREATE STREAM quotes_amt_5sec (
               symbol,
amount,
               qtime ORDERED) AS
SELECT symbol, price*volume, qtime
FROM quotes <VISIBLE '5 seconds'
             ADVANCE '5 seconds'>;
```

2. *Continuous subqueries*

A subquery that yields a stream can be embedded in the FROM clause of a continuous query, in a similar manner to the way that relational subqueries can be contained in the FROM clause of a snapshot query. When the execution of a continuous query with a FROM-clause subquery begins, the system essentially creates an anonymous derived stream for the subquery, and then replaces the FROM-clause subquery with a reference to the newly-created derived stream. The critical difference between embedding a streaming subquery and referencing a derived stream is that the subquery is evaluated only as long as its parent continuous query is active.

2. Window-based stream intervals

As described in Section 3.5, the output of a continuous SPJA block will contain the cq_winid row property that identifies the window to which each tuple belongs. To simplify composition, we allow stream intervals to be specified in units of "windows", in addition to the stream interval types described in Section 3.2.2. This syntax is equivalent to expressing a time-based stream interval on the cq_winid implicit attribute. Window-based StoRs allow windows to be preserved across SPJA blocks, and allow composition to be done at a higher level of abstraction.

Example: The following query calculates the maximum price for each symbol every 5 seconds and outputs the results over the last 1 minute interval every 5 seconds.

```
SELECT symbol, max_price
FROM
  (SELECT symbol, max(price) AS max_price
  FROM quotes <VISIBLE '5 seconds'
              ADVANCE '5 seconds'>
   GROUP BY symbol)
      v_5sec <VISIBLE 12 WINDOWS
                                        ADVANCE 1 WINDOWS>;
```

3. Output order

In this section, we discuss the ordering properties that are satisfied by the output of a continuous query. This is particularly relevant when the output of a continuous query is used as the input to another continuous query.

The output of a continuous query is always consistent with the order described by the hidden cq_winid attribute (see Section 3.5). As described in Section 4.2, this is the most frequently-used ordering that is applied to the output of a continuous query.

When the cq_open and cq_close hidden attributes are defined (that is, when the StoR utilizes a time-based interval), the output of a continuous query will also be ordered by these attributes.

Care should be taken before using the cq_rowid attribute of a continuous query as an ordering attribute. Recall that cq_rowid is defined by the order in which rows arrive at a StoR operator. While TruSQL defines the order in which windows are produced by a continuous query, the order of rows *within* a window may be undefined. This is because SQL does not specify an ordering for SQs that do not contain an ORDER BY. Therefore, while cq_rowid can be used to order the output of a continuous query, the cq_rowid that will be assigned to individual rows within a window is nondeterministic unless an ORDER BY clause is specified.

5. INTEGRATING HISTORICAL DATA

In the past few sections we limited our focus to processing CQs over "live" streams of data. In this section, we turn to the interactions between CQs and tables that contain stored historical data. One important example of this aspect of TruSQL is the "mixed" join in which streams and tables are used together in a single query. For example, the query (from Section 3.3) that compares current stock prices to historical ones:

:

```
SELECT q.symbol, (q.price – t.max_price) as delta
```

```
FROM quotes AS q <VISIBLE '10 seconds'
                   ADVANCE '10 seconds'>,
     last_year_data AS t
WHERE q.symbol = t.symbol;
```

Mixed joins are one obvious benefit of the SR model, but the unified approach supports even deeper interactions between streams and relations. In this section, we first describe how to use TruSQL to make the results of CQs accessible to clients that pose SQs. Next, we show how TruSQL can treat a table and its modifications as a stream. Finally we address the crucial issue of transactional semantics for SR systems by defining an isolation model for continuous queries.

1. Stream data adapters

The results of a CQ are often relevant to traditional clients of an SRDBMS for purposes such as accelerated reports as well as integration with legacy applications. We now discuss a flexible architecture for storing CQ results into tables. In this arrangement, there is a data producer that is driven by a continuous query, as well as a data consumer that stores the data produced by a CQ into a database table. To direct the output of a derived stream into a table, we define a new database object known as an *adapter*:

```
CREATE ADAPTER adapter_name
CONSUME stream_name
APPEND INTO table_name;
```

As each new window of the derived stream is produced, it is atomically inserted into the specified database table.[5] The target table is otherwise unremarkable, and is created via a standard DDL statement (`CREATE TABLE`), it can be queried as normal by traditional database clients, and it can be incorporated into mixed joins. Database clients can supplement, delete or modify the rows inserted by a CQ adapter, subject to the access control and concurrency control subsystems of the SRDBMS.

By separating the producer of query results from the target table into which those results are inserted, we allow the data flow from streams to tables to be adjusted in a flexible and dynamic manner.

Some database clients may only be interested in examining the most recently produced results of a continuous query. To support this scenario, we define *replace-mode adapters* that only contain the most recent windows produced by a query:

```
CREATE ADAPTER adapter_name
CONSUME stream_name
REPLACE INTO table_name
KEEP k WINDOWS;
```

As new windows are produced by the associated stream, they are inserted into the specified table; if more than *k* windows are contained in the table, the least-recently inserted window is removed. As with append-mode adapters, windows are added and removed from the target table of a replace-mode adapter in an atomic fashion.

A single producer (derived stream) can be the input source to multiple CQ adapters. A single table can be the target (consumer) of multiple CQ append-mode adapters, provided that the schemas of the streams associated with each of the adapters are compatible with the schema of the target table. Allowing a single table to be the target of more than one simultaneous replace-mode adapters is feasible, but is not addressed in this paper.

2. Streamification of tables

In a traditional RDBMS, the value of a relation can change over time as data is added, removed, and updated. In an SRDBMS, it is natural to view this sequence of state changes as a stream of events. In this section, we describe how an SRDBMS can represent the sequence of DML operations affecting a relation as a stream that

can be accessed by continuous queries. We refer to the process of creating a stream from the changes made to a relation as *streamifying* the relation.

In this paper, we make two simplifying assumptions:

- Streamification can only be performed on from modifications made to tables, not views or arbitrary relational subqueries.
- Streams can only be produced from the effects of INSERT statements, not UPDATEs or DELETEs.

We leave generalizing this feature to encompass a wider range of relations and DML operations for future work – this problem is related to the problem of capturing stream updates in general.

To create a new stream from a relation, we use another variant of the `CREATE STREAM` DDL statement:

```
CREATE STREAM stream_name (…)
AS STREAMIFY table_name ON attr , […]
ON INSERT [ WITH INITIAL CONTENTS ];
```

Each time a DML operation is executed, the value of the input relation changes from Rn to Rn+1; we capture this change of state as zero or more records in the output stream. We only produce stream events when the DML operation is guaranteed to have occurred (that is, when the transaction that executed the operation has committed).[6]

If `WITH INITIAL CONTENTS` is specified, the first records in the stream are the current contents of the table. Otherwise, the stream is initially empty.

As described in Section 2, a fundamental difference between a stream and a relation is that the former can be ordered by one or more attributes. Therefore, to move data from a table to a stream, we must define how the records in the table will be ordered when inserted into the stream. This is specified via the `ON` clause. When producing the initial content of the table on the stream, the records are inserted by sorting according to the ON keys. If a subsequent INSERT appends multiple rows to the table simultaneously, those rows are inserted into the stream according to the same ordering. Note that we require that the ordering specified by the ON clause agrees with the actual order into which new rows are added to the table.

3. Isolation for continuous queries

When a continuous query accesses one or more relations, an important consideration is how changes in those relations should be reflected in the results of the continuous query. Traditional database concurrency control techniques are insufficient, because they attempt to establish an invariant snapshot of database state for the lifetime of a query. Adopting this approach would require that modifications to stored relations never be made visible to CQs, which is clearly undesirable. In this section, we propose *window isolation* as a mechanism for exposing changes in database state to a continuous query[6]. We begin by describing *strong window isolation*, an isolation model for continuous queries that yields consistent query results while also promptly reflecting updates to input relations. We then discuss a relaxed model, *weak window isolation*, which defines a looser notion of consistency but allows a more efficient implementation.

1. *Strong Window Isolation*

In *strong window isolation*, each relation produced by the CQ's StoR operator is considered a "unit of isolation." For the duration of a single window, all the operators in the RtoR chain are guaranteed to see the same version of database state. If we consider the SQ that is formed by removing the StoR and RtoS operators from a CQ, strong window isolation is analogous to ensuring that all the operators in the SQ see a consistent snapshot of the database. Note that as described in Section 4.1, derived streams are evaluated independently from any continuous queries that reference them. It follows that the snapshot used by a derived stream advances independently from the snapshot used by any continuous queries that reference that derived stream.

When the StoR produces a new relation, the database snapshot associated with the query may be refreshed. If that occurs, any data modifications made by transactions that committed since the snapshot refresh will be visible to the CQ. This is analogous to re-evaluating the SQ contained within the CQ with a new snapshot, as in the

READ COMMITTED isolation level specified by SQL-92. Note that this model specifies what a query *must not see* (two differing snapshots within the same window), rather than guaranteeing that all database updates will be visible at window boundaries. If there is a significant cost associated with refreshing a query's snapshot (e.g. the invalidation and recomputation of cached results for relational operators), the implementation may choose to amortize this cost over multiple window boundaries. Choosing when to invalidate and recompute a cached relational operator is similar to materialized view maintenance.

2. *Weak Window Isolation*

Strong window isolation provides a strong notion of consistency, and can be implemented efficiently for chunking windows, but it inhibits commonly-applied optimizations for sliding and landmark windows. To illustrate this, consider a query of the form:

```
SELECT symbol, sum(volume)
FROM trades <LANDMARK
            ADVANCE '10 seconds'>
```

While a thorough discussion of implementation techniques is beyond the scope of this paper, we observe that each of the windows emitted by the landmark StoR in this query is a superset of the previous relation. Rather than recomputing the RtoR operator from scratch for each such relation, it would be far more efficient to retain the output relation of the RtoR operator for the previous window and then the incrementally update those results to reflect any tuples that have arrived since the last window emitted by the landmark StoR. A similar optimization can be applied to sliding windows; in both cases, there are tuples that belong to multiple windows and for which the system can avoid recomputing the RtoR operator[3][9].

However, this optimization is incompatible with strong window isolation: if a new database snapshot is used for a window, the entire result set must be recomputed from the input data, to satisfy the constraint that all tuples within a window are computed using the same database snapshot.

To allow this optimization to be applied, we propose *weak window isolation*. In this model, we divide the output relations of a StoR into a sequence of *disjoint* relations. For chunking StoRs, these disjoint relations are identical to the output of the StoR itself. For sliding and landmark windows, the output relations of the StoR can be reconstituted by combining two or more of these disjoint relations.

In weak window isolation, we guarantee that all the tuples in a single disjoint relation will use the same database snapshot. Unlike in strong isolation, in weak isolation it is possible that two or more of the disjoint relations that compose a window may be computed with different database snapshots. Although this provides a weaker notion of consistency, it has two benefits:

- The output of the RtoR operator for a given disjoint relation can be cached and used in multiple windows, allowing significant opportunities for optimization
- In some cases, the semantics provided by weak window isolation may actually be preferred by application developers. For example, a landmark window may run for a very long period of time (weeks or months). The application might prefer to have each of the disjoint output relations of the landmark StoR be computed against the most recent state of the database at that time.

While the focus of this paper is not on implementation techniques, we note in passing that efficiently implementing window isolation benefits from a concurrency control scheme in which "writers do not block readers" – in this model, a continuous query is essentially a long-running read-only transaction. In TruCQ, we leverage the MVCC scheme implemented by PostgreSQL to achieve both good performance and consistent query results.

6. ADVANCED FEATURES

This section briefly describes several features which are necessary for an industrial strength streaming engine to deal with real life problems. Many of them are time-related.

1. Composite ordered constraints

The ordering constraint in CREATE STREAM is not very different from other constraints, such as primary key or uniqueness except that it is not defined in the SQL standard. As with other constraints, it is natural to be able to declare an ordering constraint on multiple attributes, which is particularly useful when time spans several attribute domains, e.g.

```
CREATE STREAM weather (date date,
time time, temperature int,
humidity int,
ORDERED (date, time));
```

In the above example, time is measured *relative* to date and is periodic (24 hours). It is easy to imagine a non-periodic situation:

```
CREATE STREAM weather (month int,
day int, temperature int, humidity int,
ORDERED (month, day));
```

Day is measured relative to month but since the number of days in a month varies, there is no periodicity. However in both examples, the range of possible values for the subordinate domain is well known a priori. This may not be the case in general. For example, assuming that trip distance is measured from 0 each day, consider the definition:

```
CREATE STREAM travel (date date,
distance int, speed int,
ORDERED (date, distance));
```

Distance traveled is relative to the date but it may vary daily in unpredictable limits.

A different flavor of ordering by two attributes occurs when, instead of relative distance, cumulative distance is assumed in the same stream definition. For efficient implementation it is important to know the ranges of the ordering attributes. For queries, the window characteristics can be specified in terms of either ordering attribute or their combination, e.g.

```
SELECT max(temperature) AS high,
       min (temperature) AS low
FROM weather <SLICES time '3 hours'>;
```

Since `time` is the last attribute in the constraint, its name can be omitted. A combination of ordering attributes may appear in the window clause in a more complex form, e.g.

```
SELECT max(speed) AS high,
       min(speed) AS low
FROM weather <SLICES date '1 day' OR
                     distance '100'>;
```

If the ordering attribute is not numeric, concise expression of the chunking interval may be a challenge, but the following choices are always available for any type: `<SLICES 10 ROWS>` or `<SLICES 10 DISTINCT>`.

Note that, for sliding windows, it makes most sense for the finest resolution attributes of window size and window advancement to be the same.

2. Changing the order

Often, in order to solve a particular problem, it is necessary to change the ordered constraint within the context of a streaming query or subquery. Such a change may or may not influence the order of events in the stream. Appending an attribute to the constraint or removing the last attribute in the constraint does not affect the order of the events but changes *simultaneity sets*. The latter is defined as the set of events with the same values of ordering attributes. Events within such a set are logically simultaneous even though they may be internally distinguished by their insertion order or in any other way beyond the user control. Replacement of the ordering constraint

attributes may also preserve the order of events and is often required in real life problems. This motivates the following definition.

Consider some order relationship. Its *strict warping* (or *refinement*) is a subordinate order relationship: if two tuples have been in $<$ relationship with respect to the original order, they remain in $<$ relationship in the new order. Appending another attribute to the ordered constraint produces a refinement of the original order. Simultaneous events with respect to the new order are subsets of the existing simultaneity sets. *Non-strict warping* preserves $\leq$ relationship, but not necessarily $<$ relationship. Removal of the last constraint attribute typically results in non-strict warping. Another typical example is transition from time measured in months to time measured in weeks. Two dates may belong to different months, but to the same week. The underlying order of events does not change but new simultaneity sets may go across existing ones. Since there is no need to re-sort, both warpings can be efficiently handled by an implementation.

Often it is necessary to redefine the order constraint in a derived stream or (sub)query. A derived stream is a named entity with its own definition, so the constraint is simply redeclared.

For queries, ordered constraint redeclaration is an operator and syntactically becomes part of the window clause.

3. Non-contiguous time

Human beings did a good job making time related logic complex by inventing various calendars. Many industries operate on the basis of "business days". Thus, it is natural for them to specify windowing parameters in terms of the business days. A typical query may look like this: what is the largest purchase within the last 3 business days?

If events in the stream are stamped with the usual clock timestamps, complex calendar logic may be necessary to properly determine the window boundaries. Time for such applications logically flows in a non-contiguous manner, with gaps. If today is Tuesday, the last three business days are Thursday, Friday and Monday. How does one communicate within a streaming query that the time used has gaps? The most consistent way of achieving this goal is through support of non-contiguous time interval types.

Dealing with non-contiguous time involves defining a new time interval type with its own calendar arithmetic[7]. An example of a query using non-contiguous time is

```
SELECT max(amt)
FROM sales <SLICES '3 days'::bd_interval>;
```

Non-contiguous time may have gaps at any resolution. For example, within a business day, one may have working hours from 9am to 5pm, and so on. Calendar arithmetic then has to ensure that intervals over 8 working hours in length take us into the next business day, etc. Streaming database implementation must be able to handle non-contiguous time types of arbitrary complexity if that is necessary for solving a business problem.

4. Hierarchical partitioning

Partitioning is used to apply windowing logic separately to subsets of events grouped by partitioning attribute rather than to all events. It is assumed in what follows that partitioning attributes do not participate in the ordering constraint[8].

A query with partitioning would be producing windows for each partition. There is often a need to either consolidate windows across some of the partitions or to further partition the resulting windows.

Consider a stream of quotes on a set of assets from a variety of sources, organized so that each source sends an update on its quote for a given asset and side (bid or offer). When a new quote from a source arrives, one may want to republish the entire "book" of average quotes by source for the corresponding asset and side. This can all be expressed as follows:

```
SELECT asset, side, source,
       avg(price) AS price
FROM quotes <PER asset, side, source
             ON side SLICES 1 ROWS>
```

```
GROUP BY asset, side, source;
```

The order of the partition attributes is important. They induce a hierarchy in which leaves correspond to individual partitions. The "on" clause refers to the level in the partitioning hierarchy, which determines the leaves to be consolidated when one of the partitions advances. The GROUP BY clause applies to each partition before consolidation.

In the above example, for each received quote, the query output is triggered for a triple of asset, side and source, however it is consolidated over the entire set of sources for the corresponding (asset, side) combination.

7. RELATED WORK

A myriad of stream processing systems have been proposed, both by academia and by industry. In this section, we describe the relationship between some of these projects and our current work.

The query language proposed by this paper is most directly related to the Continuous Query Language (CQL), implemented by the Stanford STREAM data manager[2]. Our proposed query model retains the StoR, RtoR, and RtoS arrangement introduced by CQL, and our general language philosophy is similar to CQL's: we have endeavored to remain consistent with the "spirit" of SQL while introducing a conservative set of new language features to facilitate stream processing.

In contrast to the query model, the Truviso TruCQ system that implements TruSQL is closely related to the TelegraphCQ stream data management system developed at UC Berkeley[5].

Another notable academic streaming system is Aurora[1]. In Aurora, queries were not specified in a declarative query language such as SQL. Instead, users constructed query plans directly by connecting operators via data flows in a network model, aided by a graphical user interface.

In recent years, several industrial stream processing systems have also been developed, including Coral8[7], Esper[8], and StreamBase[12]. Most of these systems allow queries to be defined in a dialect of SQL, extended with facilities for stream processing. A key difference between these systems and the current work is TruSQL's emphasis on seamless integration between historical and streaming data, and support for the full SQL language specified by the SQL standard.

8. CONCLUSIONS

Stream query processing is a classic disruptive technology that has the potential to recast the cost tradeoffs of traditional relational query processing. We have developed the Truviso stream processing system based on the central premise that stream processing queries should be expressible in a language that is a superset of SQL. In this paper, we have described TruSQL, our Stream-Relational extension to SQL. The advantages of the SRDBMS approach are many, and include the ability to preserve the benefits of traditional RDBMS systems in terms of both technology and business considerations. Streaming systems built using an SR approach can be seamlessly integrated into existing Information Technology infrastructures while providing the extreme scalability and performance demanded by modern data-intensive applications. In this paper we have detailed the extensions we have made to SQL that enable SRDBMS functionality. We believe that these extensions can serve as a useful model for industry efforts towards seamlessly adding stream processing to standard SQL.

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 Journal* (12)2: 120-139, August 2003.
2. Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, (15)2, June 2006.
3. Arasu, A., Widom, J. Resource Sharing in Continuous Sliding-Window Aggregates. *Proceedings of the 30th International Conference on Very Large Databases (VLDB 2004).*Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S. The Object-Oriented Database System Manifesto. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*.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 the 2003 Conference on Innovative Data Systems Research*.Conway, N. Transactions and Data Stream Processing. http://neilconway.org/docs/thesis.pdf, April 2008.Coral8 Inc. http://www.coral8.comEsper. http://esper.codehaus.orgKrishnamurthy, S., Wu, C., Franklin, M. On-the-fly sharing for streamed aggregation. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*.PostgreSQL. http://www.postgresql.orgStonebraker, M., Rowe, L., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P., Beech, D. Third-Generation Database System Manifesto. *SIGMOD Record*, 19(3), September 1990.StreamBase Inc. http://www.streambase.comTruviso, Inc. http://www.truviso.com

Submitted for Conference Review May 6, 2008

---

[1] While the traditional definition of a relation excludes duplicates, SQL implementations typically allow relations to contain duplicates. Therefore, in this paper we use the term “relation” to refer to multi-sets with bag semantics.

[2] `clock_timestamp()` is a PostgreSQL built-in function that returns the current system time.

[3] Allowing streams to appear in other elements of the query language is left for future work.

[4] In this respect, derived streams are similar to materialized views in a traditional RDBMS.

[5] Inserting tuples into a table may violate a database integrity constraint. If this happens, the current behavior of TruCQ is to discard all the tuples in the window currently being inserted, but other error handling strategies are possible.

[6] If the system crashes before the corresponding stream tuples have been produced, part of the effect of the transaction will be lost; in some sense, this violates the atomicity of the database transaction. We leave the idea of a 2PC-like transaction that encompasses both the originating transaction and any consuming continuous queries in an atomic block for future work.

[7] DDL syntax for creating such interval types is not included in this paper.

[8] The semantics of partitioning on an ordering attribute is the subject of a separate discussion.