IT 10 013
Examensarbete 15 hp
Maj 2010

Evaluation of the Stream Query Language CQL

Robert Kajic

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Evaluation of the Stream Query Language CQL

Robert Kajic

There are several query languages developed for data stream management systems (DSMS), CQL (Stanford), StreamSQL (StreamBase), WaveScript (MIT), SCSQL (Uppsala University), etc. This thesis is the research phase of a two-phase project where the final goal is to provide CQL support to the Super Computer Stream Query processor (SCSQ); a DSMS developed by the Uppsala DataBase Laboratory. In this paper, the main properties of CQL, the extent to which they are implemented by the Stanford STREAM project and the expressibility of the Linear Road (LR) benchmark using CQL is investigated. An overview and comparison of SQL, CQL, StreamSQL and WaveScript is also given.

Handledare: Tore Risch
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 10 013
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

A data stream management system (DSMS) is similar to a database management system (DBMS) with the difference that while a DBMS allows searching only stored data, a DSMS in addition provides query facilities to search directly in data streaming from some source(s). The argument is that traditional DBMSs are not able to efficiently, or at all, handle large amounts of streaming data and can be greatly outperformed by a dedicated DSMS [4].

In this paper, three general purpose continuous query languages will be investigated together with their corresponding DSMSs: CQL, StreamSQL and WaveScript — implemented in STREAM, StreamBase and WaveScope respectively. Special care will be given to the main properties of CQL as well as the extent to which they are implemented by the Stanford Stream Manager (STREAM) project. CQL will be compared with SQL, StreamSQL and WaveScript.

The most widely used performance evaluation tool for DSMSs is the Linear Road benchmark [4]. It induces a set of demanding queries based on a large scale city traffic simulation. The benchmark will be introduced as well as its expressibility using CQL investigated.

2 Data Stream Management Systems and LR

The following sections will present the motivations behind three data stream management systems and their contributions. Furthermore, the Linear Road benchmark will be introduced. It simulates a city with a number of motor expressways where vehicles are subject to variable tolling.

2.1 Data Stream Management Systems

2.1.1 The Standford Stream Data Manager

The Standford Stream Data Manager (STREAM) was a project at Stanford university with the goal of developing a DSMS capable of handling large volumes of queries in the presence of multiple, high volume, input streams and stored relations [9].

The project produced a DSMS prototype and created CQL 3.1 — a declarative query language based in SQL — for expressing continuous queries on streams. The fully functional prototype is available for download on the STREAM homepage [13].

2.1.2 StreamBase

StreamBase [16] is a commercialization of the Aurora project [1] — a joint DSMS research project at Brown University, Brandeis University and Massachusetts Institute of Technology (MIT). Today, StreamBase is one of the most widely used and recognized commercial event processing platforms, offering a DSMS server and an integrated development environment aimed at rapid development of stream processing applications [18]. According to StreamBase, the StreamBase DSMS offers "the fastest performance, with the lowest latency and highest throughput" [19]. However, the company has not made any benchmarks publicly available, not of LRB nor any other DSMS benchmark. These claims are thus not easily verifiable.

StreamBase has two distinct query languages. The first, StreamSQL, is text based and has many syntactic and semantic similarities with CQL and SQL languages in general. The second, called EventFlow, is a graphical language where queries are constructed, executed and debugged using a click-and-point interface as seen in figure 2.1.2. While the languages are conceptually very different they are equal in their expressiveness [18].

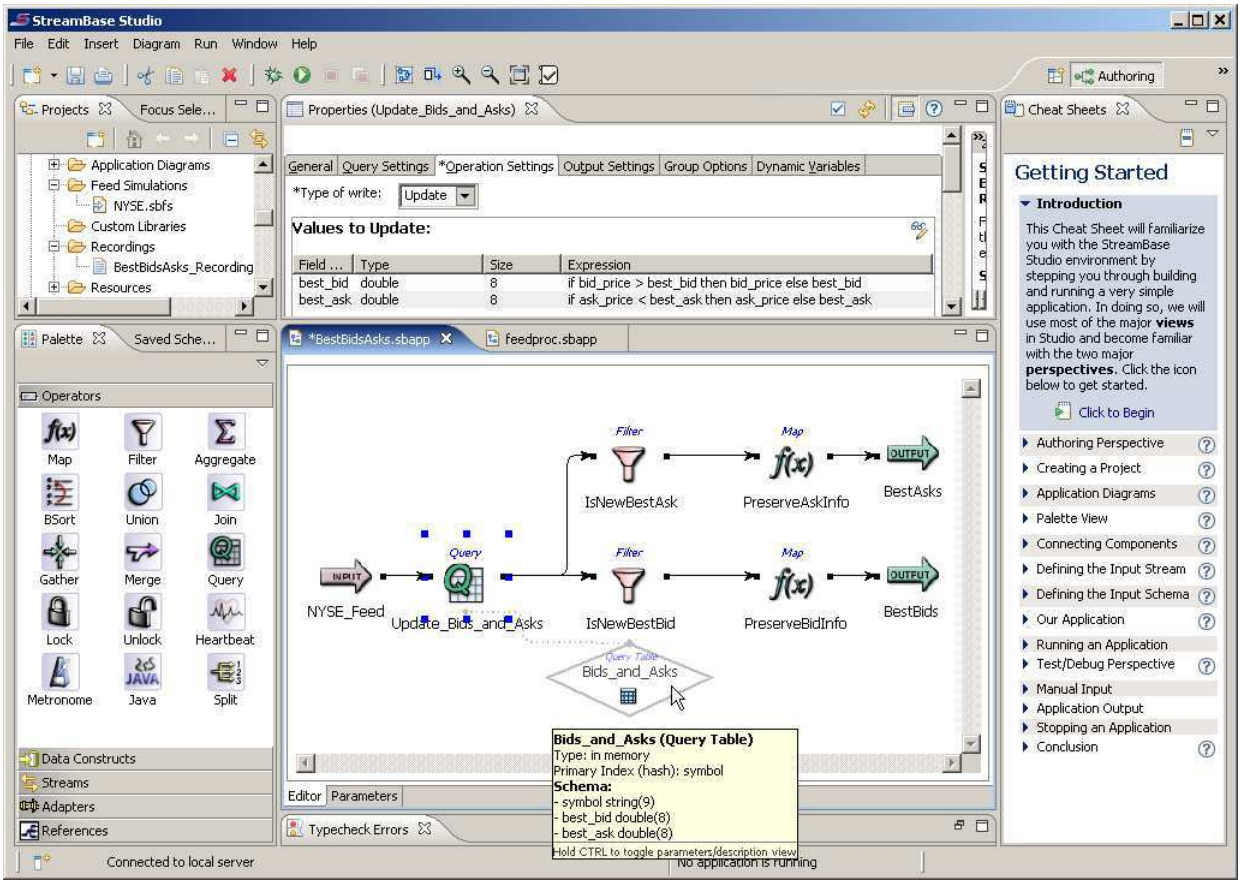

Figure 1: The StreamBase Studio is build upon the popular Eclipse IDE. Here can be seen the EventFlow query authoring interface.

2.1.3 WaveScope

WaveScope is a MIT research project in the field of high data-rate wireless sensor networks [21]. One of its contributions, most relevant to this paper, is the development of a declarative functional language for stream and signal processing called WaveScript 3.3.

2.2 Linear Road Benchmark

Congested roads during rush hours is an ever increasing problem in and around big cities. One method of alleviating this problem is through the use of variable tolling. Tolls are based on the time of day [7] and/or the current traffic situation in each vehicles vicinity (congestion, accidents, etc.). The

basic idea is to discourage use of highly congested roads and to make roads with excess capacity more attractive.

The most widely used benchmark for measuring DSMS performance is the Linear Road (LR) Benchmark (LRB). LRB simulates a fictional city with a number L of expressways where tolls are determined through variable tolling [4]. In LRB, a DSMSs performance, its *L-rating*, is determined by how many simultaneous expressways it can handle while producing timely and correct query results.

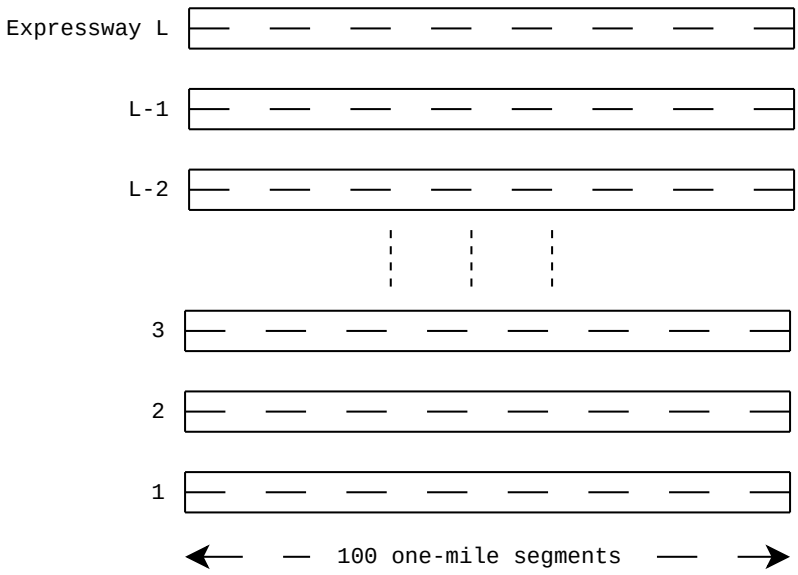

Figure 2: LRB city with L expressways, each being one-hundred miles in length.

Each expressway is 100 miles long and made up of eight lanes, four of which go from west to east, and the other four from east to west. Additionally, three of the lanes in each direction are traveling lanes with the fourth lane being an entrance and exit lane. Each expressway consists of 100 one mile segments with an entrance ramp at the start of, and an exit ramp roughly at the end of, each segment.

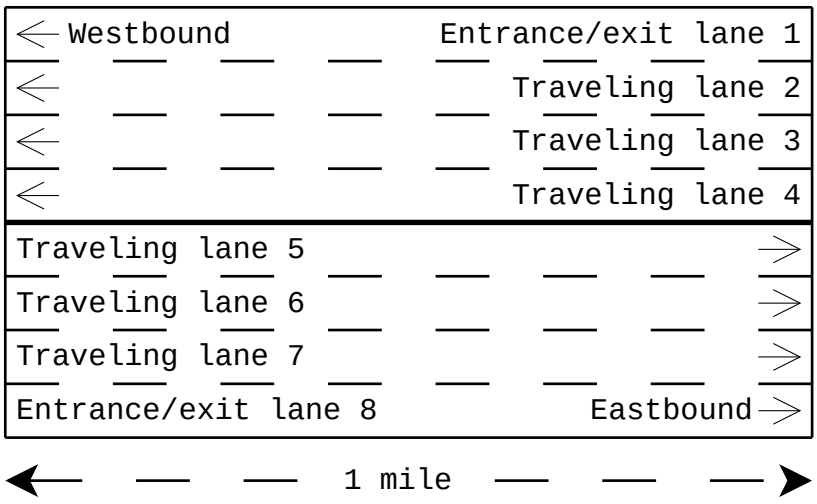

Figure 3: Each segment is one mile long, having four eastbound and four westbound lanes. Three are traveling lanes and the fourth an entrance/exit lane.

2.2.1 Notifications

Vehicles report their positions to the tolling system with an interval of 30 seconds. Whenever a vehicle sends a position from a new segment the tolling system must notify the vehicle of the cost of traveling that particular segment. The customer may choose to leave the segment by the exit ramp to avoid paying the segment toll, or continue into the subsequent segment to receive the next quote.

If two or more vehicles report the same position in four consecutive reports the tolling system must recognize this as an accident and notify vehicles in adjacent upstream segments so that they may avoid the accident.

2.2.2 Historical Queries

The tolling system must also respond to the following historical queries which are issued by the vehicles:

Account balance A vehicles current account balance in the tolling system.

Daily expenditure A sum of all tolls charged today on some given expressway.

Travel time estimation An expected travel time based on statistics of previous travel times.

2.2.3 The Simulation

LRB simulates three hours of traffic which generates roughly the following input from each expressway [4]:

1. 12 million position reports
2. 60000 account balance queries.
3. 12000 daily expenditure queries.

The following output is expected from the tolling system:

1. 2 million tolls alerts.
2. 28000 accident alerts.
3. One response for each historical query.

3 Continuous Query Languages

DSMS queries are different from conventional database queries in, e.g. SQL, where a query requests data from tables stored in the database. The result of a DSMS query can be not only a set of tuples as in SQL, but also a potentially infinite stream of tuples. Furthermore, stream queries are continuous queries (CQs) in that they run indefinitely, or until they are terminated, while conventional queries are executed on demand and run until all requested data is delivered. In this paper, I will take a closer look at CQL 3.1 and two related stream processing languages: StreamSQL 3.2 and WaveScript 3.3.

3.1 CQL

Syntactically CQL is very similar to the *SELECT* statement of SQL making it easy to learn and understand for users with previous experience of SQL-like languages. Furthermore, being a declarative language, it leaves all choices of how to execute and optimize the query to the DSMS.

A dominating part of the data manipulation of a CQL query is performed by relation-to-relation operators [3, 2, 22]. This approach was chosen so that well understood relational concepts could be reused and extended. The operators include many of those normally found in SQL, such as projection, selection, aggregation, joining, grouping, etc.

Additionally, CQL has stream-to-relation and relation-to-stream operators which, as their names suggest, convert from streams to relations and vice

versa. Together with the relation-to-relation operators they offer great flexibility in how data can be manipulated; once a stream-to-relation operator has been applied to a stream it can be subjected to regular relation-to-relation operators after which it may be, if necessary, transformed back to a stream using a relation-to-stream operator.

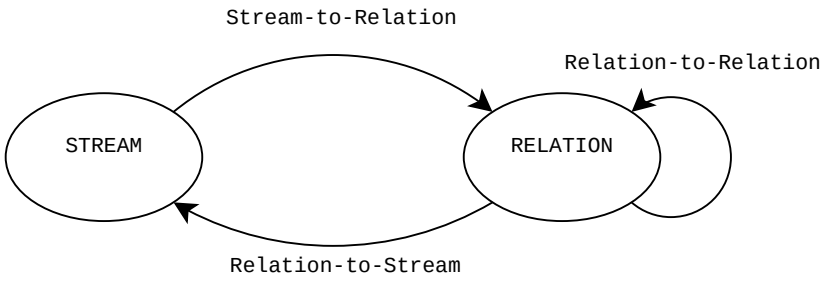

Figure 4: Streams are converted to relations using CQLs sliding window operators. Relations are manipulated using standard relational operators and can be converted back to streams using one of the relation-to-stream operators available in CQL.

3.1.1 Stream-to-Relation Operators

The stream-to-relation operators in CQL are based on a *sliding window* [5] which can be though of as view upon a stream that, at any point in time, reflects the viewed part of the stream as a relation. As time flows the window moves over the stream and the contents of the relation are changed to reflect the current view.

To express sliding windows CQL uses a window specification language inspired by SQL-99. The available window types are: time based 3.1.1.1, tuple based 3.1.1.2 and partitioned windows 3.1.1.3. In the following paragraphs, each will be described through a series of examples.

3.1.1.1 Time Based Windows

Given the following streams:

```
Auctions(auction_id, seller, time)
Purchases(auction_id, buyer, cost, time)
```

Listing 1: Each auction tuple contains an integer *auction_id* auction identifier, an integer *seller* which identifies the seller and a timestamp *time*. Each purchase tuple contains an integer *auction_id* auction identifier, an integer *buyer* which identifies the buyer, an integer *cost* for the auction ending price and a timestamp *time*. In both streams, the timestamps denote when the associated tuple was emitted.

The question "What is the total amount spent by Luke on auctions from John, during the last day." can be expressed using the following CQL query:

```
SELECT SUM(P.cost)
FROM Auctions AS A, Purchases [RANGE 1 DAY] AS P
WHERE A.auction_id = P.auction_id
  AND A.seller = "John";
  AND P.buyer = "Luke";
```

Listing 2: Time based sliding window.

The query uses *[RANGE 1 DAY]* to define a time based sliding window on the Purchases stream, which produces the continuously updated relation R. At any time T the relation R will contain all stream elements in the Purchases stream with timestamps ranging from T and going back one day. As time passes, the window moves forward, excluding purchases as they become too old and including newly made ones. The current contents of the window will always be reflected in the relation R.

The aggregate function *SUM()* is continuously applied on the current contents of the relation R producing the final result of the query.

3.1.1.2 Tuple Based Windows Using the same input streams, the question "What is the total amount spent by Luke on his latest ten purchases from John." can be expressed using the following CQL query:

```
SELECT SUM(P.cost)
FROM Auctions AS A, Purchases [ROWS 10] AS P
WHERE A.auction_id = P.auction_id
  AND A.seller = "John";
  AND P.buyer = "Luke";
```

Listing 3: Tuple based sliding window.

The query differs from the previous one in that we now use a tuple based

window. *ROWS* is similar to *RANGE*, with the difference that while *RANGE* defines a window based on time, the *ROWS* operator expects a physical window size *N*, i.e., a maximum number of tuples that may be included in the sliding window.

3.1.1.3 Partitioned Windows The following CQL query expresses "Take the 10 most recent auctions from each seller and return the seller of the most expensive item.":

```
SELECT A.seller, MAX(P.cost)
FROM Auctions [PARTITION BY A.seller ROWS 10] AS A,
     Purchases AS P
WHERE A.auction_id = P.auction_id
GROUP BY A.seller;
```

Listing 4: Partitioned sliding window.

The partitioned window operator expects a stream *S*, a positive number of tuple attributes and an integer *N*. It splits the stream *S* into sub-streams such that the given attributes are unique for each stream. A tuple based window of size *N* is then applied on each sub-stream to create a number of continuously updated sub-relations. Finally, a relation *R*, i.e., the query output, is formed by taking the union of all sub-relations.

3.1.2 Relation-to-Stream Operators

In the previous examples the results of the queries have all been continuously updated relations. The following relation-to-stream operators are available when it is desirable for a query to return a stream:

ISTREAM(R) Defines a stream of elements *(r, T)* such that each r is in relation R at time T, but was not in R at time *T-1*.

DSTREAM(R) Defines a stream of elements *(r, T)* such that each r was in relation R at time *T-1*, but is not in R at time T.

RSTREAM(R) Defines a stream of elements *(r, T)* such that each r is in relation R at time T.

As an example, the following CQL query expresses "A stream of sellers of the most expensive sold item, taking into account no more than 2000 of each sellers most recently sold items.":

```
ISTREAM(SELECT A.seller, MAX(P.cost)
        FROM Auctions [PARTITION BY A.seller ROWS 1000] AS A,
             Purchases AS P
        WHERE A.auction_id = P.auction_id
        GROUP BY A.seller);
```

Listing 5: Using the *ISTREAM* operator to produce a stream of the most expensive items.

The query will produce a new tuple (<seller, max>, T) whenever a seller achieves a new maximum at time T, as compared with *T-1*.

3.1.3 Processing of tuples

A fundamental property of CQL is that windows are calculated (conceptually) using a time-driven model [10]. All stream tuples are of the form *(value, timestamp)* and the contents of a window are updated at the end each time step T, with regard to tuples with timestamps equal to or smaller than T. Regardless of how many tuples arrive at any given timestamp, the window is only updated when it is known that no further tuples can arrive for said timestamp. This model be contrasted to a tuple-based model, where a windows' contents are re-evaluated every time a new tuple arrives.

To highlight one of the differences between the two models I will illustrate with an example. Given the following stream (which uses the *Purchases* schema from Listing 1):

```
Purchases(auction_id, buyer, cost, time) =
          (1, "Lars",  30, 0)
          (2, "Mia",   10, 1)
          (3, "Sven",  50, 1)
          (4, "Klara", 50, 1)
          (5, "Svea",  60, 2)
```

Listing 6: Stream of pruchases.

And using the following query:

```
ISTREAM(SELECT AVG(cost) as AvgCost, time
        FROM Purchases [ROWS 10]);
```

Listing 7: Calculate the average cost of the latest ten purchases.

We get these results:

3.1.3.1 Time-driven Using a time-driven model the window is re-evaluated only at the end of each time step, giving the following output stream:

```
(30, 0)
(35, 1)
(40, 2)
```

Listing 8: Output stream for time-driven model.

3.1.3.2 Tuple-based With a tuple-driven model the window is re-evaluated every time a new tuple arrives. This gives us the following output stream:

```
(30, 0)
(20, 1)
(30, 1)
(35, 1)
(40, 2)
```

Listing 9: Output stream for tuple-driven model.

Further examples and a very thorough investigation of the strengths and limitations of both models can be found in [10].

3.1.4 Defaults

Two of the main goals when designing CQL were that simple queries should be easy to write and do what you expect [22]. In order to achieve these goals the language has a number of syntactic defaults such as that: an unlimited (infinite) window is applied on streams by default, the relation-to-stream operator can often be omitted as *ISTREAM* is applied by default, a special *Now* window exists and is equivalent to a *[RANGE 1 SECONDS]* window, etc.

3.1.5 Implementation in STREAM

The current implementation of CQL in STREAM does not provide all features described in [3, 2, 22]. Some of the missing features can, however, be expressed if intermediate named queries are used [14]. That is, a query can be given a name which can later be referenced by another query, in place of a relation or a stream. The following features are not supported:

1. The *WHERE* clause may not contain sub queries:

```
RSTREAM(SELECT *
        FROM cars
        WHERE cars.id IN (SELECT accident.carid
                          FROM accident));
```

Listing 10: No sub queries in the *WHERE* clause.

2. The *HAVING* clause is not supported:

```
SELECT segid, SUM(carid) AS ncars
FROM segments
GROUP BY segid
HAVING ncars > 1;
```

Listing 11: No *HAVING* clause

However, it is possible to rewrite the query using a intermediate named query:

```
GroupedSegments(segid, ncars):
SELECT segid, SUM(carid) AS ncars
FROM segments
GROUP BY segid;

SELECT *
FROM GroupedSegments
WHERE ncars > 1;
```

Listing 12: Intermediate query.

3. Arithmetic with aggregations is not supported in the *PROJECT* clause:

```
SELECT segid, MAX(speed)-MIN(speed) AS speed_distance
FROM segments
GROUP BY segid
```

Listing 13: No aggregation in the *PROJECT* clause.

4. Attributes may only be of type Integer, Float, Char(n) or Byte and it is not possible to perform type casting.

5. It is not possible to specify a *SLIDE* value for windows, i.e., by how much a window should move when it moves forward.

6. The *INTERSECT* operator is not supported, but *UNION* and *EXCEPT* are.

3.1.6 Linear Road Benchmark Expressibility

The STREAM project has not published an implementation of the LR benchmark. However, they have defined a benchmark specification using CQL [12]. The specification does not include the stream of travel-time estimates, but this output stream is also skipped by all currently published LR implementations; Super Computer Stream Query processor [23], Aurora [4], Stream Processing Core [11] and XQuery [6].

The CQL-based specification published at [12] defines the following output streams:

TollStr The stream of tolls for cars entering congested segments.

AccNotifyStr The stream of accident notifications to cars which are moving toward and are in close proximity to a accident segment.

AccBalOutStr The stream of account balance notifications answering adhoc account balance queries.

ExpOutStr The stream of todays expenditures notifications answering the corresponding adhoc queries.

The streams are defined using a number of intermediate named queries. I will not present those queries here — they are readily available at [12].

3.1.6.1 The output stream *ExpOutStr*

is interesting because it makes use of a special "Today Window" to define a window which contains all stream tuples with timestamps within the current day (note it uses a deprecated relation-to-stream operator syntax):

```
SELECT RSTREAM(query_id, E.car_id, -1 * SUM(credit)) FROM
ExpQueryStr [NOW] as Q, AccTransStr[Today Window] as T WHERE
Q.car_id = T.car_id;
```

Listing 14: Using the Today Window to define the LRB output stream *ExpOutStr*.

The query is taken out of its context and you may want to take a closer look at the definitions of *ExpQueryStr* and *AccTransStr* at [12]. Shortly, *ExpQueryStr* is the input stream of adhoc queries requesting todays expenditures, and *AccTransStr* is the stream of all car account transactions.

The *ExpOutStr* should produce a stream of todays expenditures, but using CQL it is not possible to define a window with a specific "start" time, e.g. the start of the current day, nor a window slide, e.g. that the window should move by one day at the end of each day. While the suggested "Today Window" expresses exactly those requirements it is actually not implemented by the protoype [13]; i.e., the definition of the ExpOutStr stream unusable in practice. Indeed, a formal definition of the benchmark, in the form of a CQL script, can be found in the latest source code distribution of STREAM [13] (in $SOURCE/examples/scripts/linearroad); the only missing part is the *ExpOutStr* query.

3.1.7 Slide parameter

A slide parameter has been incorporated into the most recently released version of STREAM. This makes it possible to define a "Today Window" as it was described in the previous section, and to fully implement LR using CQL and STREAM. The modified source is available at [GitHub]. Here is an informal description of the semantics of the new parameter:

The slide is optional and should be specified immediately after the window operator *RANGE*. It is currently not possible to define a slide together with the *ROWS* operator. The slide is expressed using a time specification which follows the exact same syntax as *RANGE*. Finally, specifying a range smaller than the slide has undefined results.

3.1.7.1 Simple example The following example defines a window with a range of 10 seconds, and a slide of 5 seconds:

```
SELECT *
FROM S [RANGE 10 SECONDS SLIDE 5 SECONDS];
```

Listing 15: Slide of 5 seconds.

Once the window encompasses a range of 10 seconds, it will move forward 5 seconds. All tuples with timestamps between the windows' original and new position will be dropped.

3.1.7.2 Revisiting the Today Window The window *[RANGE 1 DAY SLIDE 1 DAY]* is equivalent to the "Today Window" described at [12]. It can be used to define the *ExpOutStr* stream in the following way (I am using the deprecated relation-to-stream operator syntax for consistency with the original query):

```
SELECT RSTREAM(query_id, E.car_id, -1 * SUM(credit))
FROM ExpQueryStr [NOW] as Q,
     AccTransStr [RANGE 1 DAY SLIDE 1 DAY] as T
WHERE Q.car_id = T.car_id;
```

Listing 16: The *ExpOutStr* stream using the new SLIDE parameter.

3.2 StreamSQL

With a similar reasoning as the STREAM project when creating CQL — to take advantage of SQLs ubiquity in DBMSs — StreamSQL also has its foundations laid in SQL. Therefore, StreamSQL shares many syntactic and semantic similarities with both SQL and CQL.

The language has operators which deal with two data types; streams and relations, both of which may appear in the from clause of queries. Streams are defined as potentially infinite sequences of tuples and relations are either regular relational tables or stream windows. The main distinction here, from CQL, is the addition of operators that deal directly with streams. Furthermore, StreamSQL stream windows are tuple-driven instead of time-driven. This is investigated in greater detail in sections 3.2.1 and 3.1.3.

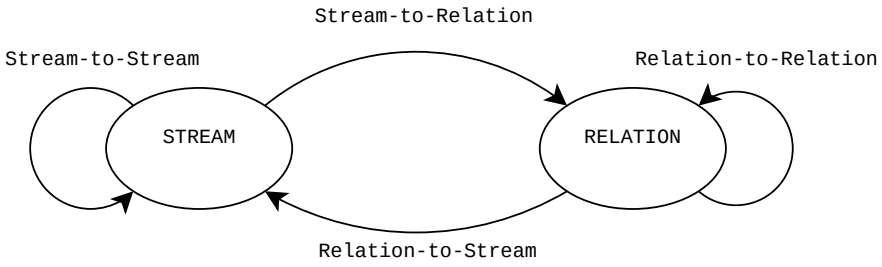

Figure 5: As in Figure 3.1, streams are converted to relations using sliding window operators. Relations are manipulated using standard relational operators and then converted back to streams. Additionally, in StreamSQL, streams can be directly manipulated, whiteout intermediate conversions, using a set of stream-to-stream operators.

3.2.1 Tuple-driven Processing

In section 3.1.3 I touched upon how a continuous languages' tuple processing model can influence the behaviour of its operators and consequently the output produced by its queries. StreamSQL uses a tuple-driven model ordering tuples not only on their timestamps, but also on their order of arrival [5].

Tuples are assigned ranks based on arrival and processing is performed in rank-ascending order as far as possible, i.e., a tuple with rank R is processed before all tuples with $rank > R$.

3.2.2 Stream-to-Stream Operators

This section will give an introduction to some of the available stream-to-stream operators in StreamSQL. For a complete reference of all available operators please refer to the StreamSQL documentation [20].

Given the following Auctions stream:

```
Auctions(auction_id, seller, starting_price, time)
```

Listing 17: Each auction tuple contains an integer *auction_id* auction identifier, an integer *seller* which identifies the seller, an integer *starting_cost* of the initial auction price and a timestamp *time* of when the tuple was emitted.

3.2.2.1 Filter To filter the Auctions stream and keep auctions with a starting price bellow 10 we could use the following query:

```
SELECT *
FROM Auctions
WHERE starting_price<10;
```

Listing 18: Stream filtering.

The query will output a new stream from which overly expensive items have been excluded.

3.2.2.2 Stream Join A stream can be joined with static tables, combining each stream tuple with a set of tuples from the tables. To illustrate, I will add schemas for two tables; one that associates auctions with categories and one that contains the actual category information. This will make it possible to associate each auction with several categories.

```
AuctionCategories(auction_id, category_id)
Categories(category_id, name)
```

Listing 19: Static table schemas for auction categories and category information.

The auctions stream can then be joined with the static tables:

```
SELECT *
FROM Auctions A, AuctionCategories AC, Categories C
WHERE A.auction_id=AC.auction_id
  AND AC.category_id=C.category_id;
```

Listing 20: Stream joining.

3.2.2.3 Stream Union It is possible to combine two streams using the union operator, the result being an interlacing of the two streams with their order of arrival preserved. Given the following streams of Swedish and Norwegian auctions, where tuples arrive at *time*:

```
Auctions_Sweden(auction, time) =
                (auction, 7),
                (auction, 10);

Auctions_Norway(auction, time) =
                (auction, 0),
                (auction, 6),
                (auction, 9);
```

Listing 21: Norwegian and Swedish auctions.

The streams can be unified:

```
SELECT *
FROM Auctions_Sweden UNION Auctions_Norway;
```

Listing 22: Stream union.

To produce the following output stream:

```
Auctions_Union(auction, time) =
  (Norwegian_auction, 0)
  (Norwegian_auction, 6)
  (Swedish_auction, 7)
  (Norwegian_auction, 9)
  (Swedish_auction, 10)
```

Listing 23: Union of Swedish and Norwegian auctions.

3.2.2.4 Pattern matching StreamSQL has pattern matching operators which allow detection of temporal or range-based patterns between tuples from one or more streams. For example it is possible to detect that an auction is not followed by a bid within T seconds.

This is achieved through the *PATTERN* clause which consists of a *template* that references one or more streams and defines some relationship between those using *pattern operators*, and a *window* that defines the maximum allowed duration for the defined relationship. The window may either be time-based or value-based. I.e., with a maximum time, or a maximum range of values on some field, during which the relationship must occur.

I will illustrate with an example. Using a bids stream with the following schema:

```
Bids(auction_id, buyer, cost, time)
```

Listing 24: Each bid tuple contains an integer *auction_id* auction identifier, an integer *buyer* which identifies the buyer, an integer *cost* for the current bid price and a timestamp *time* of when the tuple was emitted.

This query then expresses "All auctions which have no bids within ten minutes.":

```
SELECT *
FROM PATTERN Auctions THEN NOT Bids WITHIN 600 TIME;
```

Listing 25: Using pattern matching to find auctions with no bids.

The template *Auctions THEN NOT Bids* defines a relationship where auctions are not followed by bids, and *WITHIN 600 TIME* specifies a time-based window during which the relationship must occur. Whenever an auction is created the template will immediately match. Additionally, if there are no bids for ten minutes the window will be satisfied and the query will output the unpopular auction. Otherwise, i.e., if a bid arrives before the window closes, the template will no longer match and the auction no longer needs to be monitored for bids.

The following figure depicts a scenario where the template discards the first auction and accepts the second:

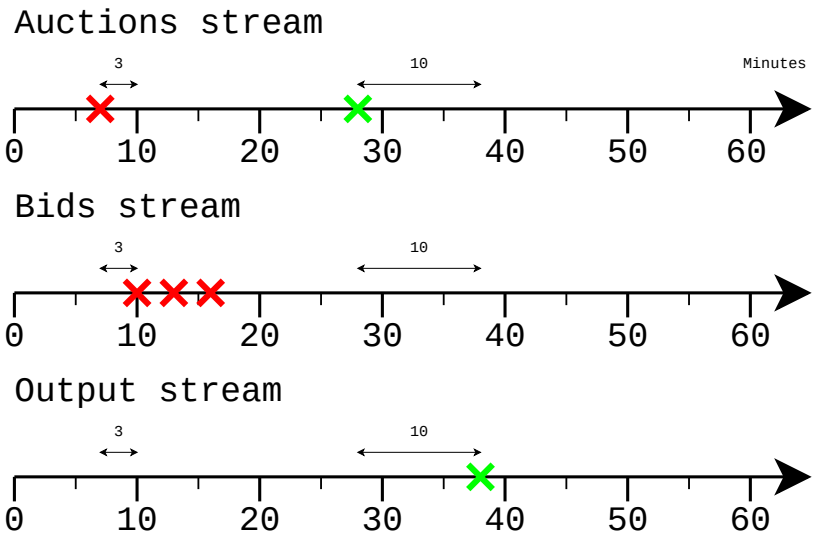

Figure 6: Equally colored auctions and bids have the same *auction_id*. The red auction is invalidated by the pattern template in Listing 25 as a bid arrives after only three minutes, i.e., within the ten minutes specified by the pattern window. The green auction, however, has no bids within ten minutes and the pattern produces the auction tuple on the output stream.

A formal definition of the pattern matching language, together with additional examples, can be found at [17]. Furthermore, the StreamBase StreamSQL webinar [15] has additional pattern matching examples and an introduction to StreamSQL in general.

3.2.3 Stream-to-Relation Operators

Like CQL, StreamSQL makes use of the notion of sliding windows 3.1.1. It is possible to define windows "inline" , i.e. as part of a query, but also using a separate *CREATE WINDOW* statement, so that a window may be used by more than one query. Compared with CQL, the StreamSQL window offers a few additional features such as:

1. Ability to define a slide.
2. An *offset* that specifies how long to wait before opening the first window on the stream.
3. A *timeout* after which the window closes.
4. The ability to define windows not only based on time, and number of tuples, but also on a range on some user defined stream field.

Disregarding the added functionality, the StreamSQL and CQL windows are semantically very similar, with an important exception. In CQL, the state of a window changes at the end of each time step, while in StreamSQL it is changed every time a new tuple arrives [5].

3.2.4 Relation-to-Stream Operators

Unlike CQL, where there are three different relation-to-stream operators available, StreamSQL has a single relation-to-stream operator with the limitation that the *PROJECT* clause must perform some kind of aggregation. The operator streams the result of the aggregation every time a new tuple arrives, instead of the end of each time step.

To give an idea of the languages syntax, the "most expensive item" query in Listing 5 is rewritten using StreamSQL:

```
CREATE STREAM MostExpensiveAuctions AS
  SELECT A.seller, MAX(P.cost)
  FROM Auctions [SIZE 10 ADVANCE 1 TUPLES PARTITION BY A.seller] AS A,
       Purchases AS P
  WHERE A.auction_id = P.auction_id
  GROUP BY A.seller;
```

Listing 26: Stream of most expensive items.

The window specification creates a window with a fixed length of 10 tuples, that advances one tuple each time a new window is opened. The output of the query is sent into a stream called *MostExpensiveAuctions*.

3.3 WaveScript

With sensors, such as microphones and cameras, that produce signals with data rates of hundreds of thousands of samples per second it is often desirable to analyze the signal in chunks, instead of splitting it into a sequence of tuples as is the usual approach by most DSMSs. Furthermore, the analysis of such signals is often application specific and must be processed by used defined code which introduces the non negligible overhead of converting the signal data back and forth from the DSMS to some external langauge such as C or MATLAB for processing.

WaveScript is a programming language developed to integrate high data-rate stream processing with signal processing where the traditional tuple-based approach of analyzing streams doesn't provide sufficient performance. The language introduces a new first-class data type called *signal segment* that groups signal samples into chunks. Each signal segment contains a fixed

number of signal samples emitted at an regular rate, allowing the segment to timestamp only the first sample (the remaining samples being implicitly timestamped), and by doing so avoiding the substantial space overhead incurred by per-tuple timestamps. A signal segment can be directly processed using native WaveScript operators which perform, e.g., signal filtering, spectrum analysis, resampling, etc. [8], or users may define new operators directly using WaveScript, by doing so avoiding data conversion to and from foreign functions defined in an external language.

4 Conclusions

In this paper, a number of continuous query languages were investigated and compared. Special attention was given to CQL, its operators and the level at which the Linear Road benchmark can be expressed using STREAMs' implementation of CQL.

CQL is an extension of the ubiquitous relational database language SQL and makes use of its relation-to-relation operators to the furthest extent possible. To do this CQL offers a number of stream-to-relation operators which, after they have been applied to a stream, allow its data to be manipulated using well understood relational operators, such as projection, selection, aggregation, joining, grouping, etc. Once the data has been appropriately transformed it can converted back to a stream using some of the relation-to-stream operators available in CQL.

The stream-to-relation operators available in CQL are based on the concept of a sliding window. The sliding window is not unique to CQL but is available in most continuous query languages as a tool for dividing the stream into finite segments which can then be readily manipulated and analyzed. To calculate its windows, CQL uses a time-driven model, as opposed to tuple-driven, which is for example used by StreamSQL. In any CQL implementation, it is important to adhere to the semantics of the time-driven model to get the same behaviour across all implementations.

CQL is not able to express the linear road benchmark in its entirety. This can mainly be affected to CQLs inability of expressing windows with a variable *SLIDE* parameter, or more specifically a "Today Window" 3.1.6.1. However, using the new *SLIDE* parameter, which was added and described in this paper 3.1.7, it should be possible to implement LRB using STREAM. STREAM includes a significant LRB implementation which comes short of being complete because it does not implement the *ExpOutStr* output stream 3.1.6.1. This implementation should serve as a natural starting point to any endeavour of implementing the complete LRB.

Compared with StreamSQL, CQL is relatively small in terms of available features. StreamSQL has all the features available in CQL, and additionally offers additional operators in all operator classes (relation-to-relation, relation-to-stream and stream-to-relation). Furthermore it has a completely new class of stream-to-stream operators which allow manipulation of streams without intermediate conversion to and from relations. These allow for stream filtering, joining, pattern matching, etc.

However, StreamSQL uses a tuple-driven model and therefore some queries expressible using CQL are not possible using StreamSQL and vice versa. Research has recently been undertaken in unifying the tuple- and time-based models [10], so that the benefits of both can be harvested and their individual limitations avoided.

WaveScript is aimed at stream applications where segments of stream tuples are more meaningfully analyzed than isolated stream tuples. By grouping tuples into segments, and treating those segments as first-class objects which are directly manipulable by the language, performance is increased by several orders of magnitude.

5 Future Work

It has been mentioned that this thesis is the first phase of a project which hopes to add CQL support to SCSQ. An important, yet unresolved issue, on the way to reaching this goal, is to determine how CQL operators can be expressed in terms of SCSQL operators (SCSQL is the query language used by SCSQ), and which, if any, CQL operators require new functionality in SCSQ.

StreamSQL shares many similarities with CQL, but has had time to develop a richer set of features and uses a tuple-driven model. When CQL support has been added to SCSQ it should be interesting to evaluate if it is possible, and if so, how to give support to StreamSQL in SCSQ. StreamSQL has many features not available in CQL, most notably stream-to-stream operators, with pattern matching in particular, which give it additional expressive power.

A recent paper [10] seeks to integrate the differences between CQLs time-based model with StreamSQLs tuple-based and proposes the new *SPREAD* operator. It should be investigated if the ideas behind the *SPREAD* operator can help in implementing CQL and StreamSQL in SCQL in a general way.

Together with its source code, the STREAM project has released an incomplete definition of the LR benchmark. Part and possibly the whole problem was CQLs inability to express windows with a *SLIDE* parameter. Now

that the *SLIDE* parameter is available, it would be interesting to complete the LRB definition in order to evaluate STREAMs performance and determine its L-value. There may be other issues, unrelated to the *SLIDE* parameter which prevent the completion of the benchmark; if so, these obstacles should be identified and resolved.

References

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. Technical Report 2004-20, Stanford InfoLab, 2004.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 480–491. VLDB Endowment, 2004.

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM.

[6] Irina Botan, Donald Kossmann, Peter M. Fischer, Tim Kraska, Dana Florescu, and Rokas Tamosevicius. Extending XQuery with window functions. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 75–86. VLDB Endowment, 2007.

[7] Center for Urban Transportation Research. Variable tolling starts in Lee County, Florida. `http://www.cutr.usf.edu/pubs/news_let/articles/winterC98/news936.htm`.

[8] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. The Case for a Signal-Oriented Data Stream Management System. In *CIDR*, pages 397–406. www.crdrdb.org, 2007.

[9] The Stream Group. STREAM: The Stanford Stream Data Manager, 2003.

[10] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming SQL standard. *Proc. VLDB Endow.*, 1(2):1379–1390, 2008.

[11] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2006. ACM.

[12] Stanford. STREAM Linear Road Benchmark Specification. `http://infolab.stanford.edu/stream/cql-benchmark.html`.

[13] Stanford. STREAM Prototype Source Code. `http://infolab.stanford.edu/stream/code/stream-0.6.0.tar.gz`.

[14] Stanford. STREAM User Guide and Design Document. `http://infolab.stanford.edu/stream/code/user.pdf`.

[15] StreamBase. SQL on Streams for CEP Webinar. `http://www.streambase.com/64f2a328-cd57-40a0-805f-86748aecd245/download.htm`.

[16] StreamBase. StreamBase homepage. `http://www.streambase.com/`.

[17] StreamBase. StreamBase Pattern Matching Language. `http://www.streambase.com/developers/docs/latest/reference/patternquery.html`.

[18] StreamBase. StreamBase Studio features web page. `http://www.streambase.com/products-StreamBaseStudio.htm`.

[19] StreamBase. StreamBase "Why StreamBase" webpage. `http://www.streambase.com/solutions-streambase.htm`.

[20] StreamBase. StreamSQL documentation. `http://streambase.com/developers/docs/latest/streamsql/index.html`.

[21] WaveScope. Wavescope project homepage. `http://nms.csail.mit.edu/projects/wavescope/`.

[22] Jennifer Widom. CQL: A Language for Continuous Queries over Streams and Relations. Slides from a talk given at the Database Programming Language (DBPL) Workshop `http://www-db.stanford.edu/~widom/cql-talk.pdf`, 2003.

[23] Erik Zeitler and Tore Risch. Scalable splitting of massive data streams. In Hiroyuki Kitagawa, Yoshiharu Ishikawa, Qing Li, and Chiemi Watanabe, editors, *DASFAA (2)*, volume 5982 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.