Large Scale Integrating Project

Grant Agreement no.: 257899

D3.2 – Query language survey and selection criteria

SMART VORTEX –WP3-D3.2

Project Number	FP7-ICT-257899
Due Date	
Actual Date	
Document Author/s:	Tore Risch, Robert Kajic, Erik Zeitler, John Lindström, Mathias Johansson, Hans-Ulrich Heidbrink
Version:	0.2
Dissemination level	PU
Status	Final
Contributing Sub-project and Workpackage	WP3
Document approved by	RTDC

Co-funded by the European Union

Document Version Control			
Version	**Date**	**Change Made (and if appropriate reason for change)**	**Initials of Commentator(s) or Author(s)**
.1	11/08/05	First version	TR, RK, EZ
.2	11/09/28	Updated first version	TR, JL, EZ, MJ, HHk

Document Change Commentator or Author		
Author Initials	**Name of Author**	**Institution**
TR	Tore Risch	Uppsala University
RJ	Robert Kajic	Uppsala University
EZ	Erik Zeitler	Uppsala University
JL	John Lindström	Luleå Technical University
MJ	Mathias Johansson	Alkit Communications
HHk	Hans-Ulrich Heidbrink	InConTec GmbH

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
V 0.2	30/09/2011		IK

Catalogue Entry

Title	Query language survey and selection criteria
Creators	
Subject	
Description	
Publisher	
Contributor	
Date	
ISBN	
Type	
Format	
Language	English
Rights	

Citation Guidelines

EXECUTIVE SUMMARY

This document gives an overview of typical query languages and systems for data stream processing. Based on the study the report ends with a recommendation for the choice of query language in Smart Vortex project.

The Smart Vortex project concerns customized continuous query processing over different kinds of streams originating from industrial contexts. Data originating in different kinds of streams need to be combined with data in regular databases, and thus it is required designing an extensible federated Data Stream Management System (DSMS) where both streaming and regular data sources can be incorporated.

Parts of the data processing will require advanced computations (e.g. statistical) made in real-time. To cope with this challenge, the DSMS must be extendible with application dependent functions called in continuous queries, and the DSMS made federated having the possibility to be extended with customizable computations. Some of the computations made in real-time may be relatively expensive to compute. Therefore a federated DSMS must have the ability to perform expensive computations for decision support in real-time over streams.

The conclusion is that the most suitable query language and system for building a federated DSMS to meet the requirements posed by the Smart Vortex project is SCSQ. The federated DSMS should be built by extending SCSQ and SCSQL with new facilities.

TABLE OF CONTENTS

1 INTRODUCTION

A central technology in the Smart Vortex project is the ability to search and analyze high volume data streams in a distributed environment with means of a Data Stream Management System (DSMS). The searches and analyzes are specified using continuous queries (CQs) expressed in a query language. In this document the principles of existing such query languages are overviewed.

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 [1].

In this document, a number of general purpose continuous query languages are investigated together with their corresponding DSMSs. The query language CQL from Stanford is used as a reference for typical continuous query language primitives. CQL is compared with SQL and the other continuous query languages.

The most widely used performance evaluation tool for DSMSs is the Linear Road benchmark [1], which introduces a set of demanding queries based on a large scale city traffic simulation.

It should be noted that there is not yet any standard proposal for a data stream query language. Many of the principles for how to define such a query language are still being debated in the scientific literature [9][10].

2 IMPORTANT CONCEPTS

2.1 Database Management System (DBMS)

A DBMS is a system software for scalable management of large data volumes usually stored on disk. DBMSs provide for languages and components to search and update databases, as well as for storing meta-data called the *database schema* about the stored data.

2.2 Query

A *query* is a request for the retrieval of some data from a database. A query is executed immediately to retrieve the desired information, for example:

> ‘What machines are currently operating at too high temperature?’

This is called a *passive query* since the system processing the query passively waits for users to send the queries for execution.

2.3 Query Language

Queries are expressed in terms of some *query language*. For example, queries to relational databases are usually expressed in the query language SQL, while queries to RDF based information models are usually expressed in SPARQL. Queries provide for users and programmers a very general way to specify data selections (filters), combinations (joins), and computations over data stored in databases..

2.4 Data Stream

A data stream is a continuous flow of tuples (events) of measurements from some artefact. A data stream can be seen as an ever growing sequence of tuples. A data stream can be live in case it is communicated directly from its source without being stored on some media. At a given point in time one can save a snapshot of the current state of a stream on disk or some other media.

2.5 Continuous Query (CQ)

Regular passive queries are not sufficient from searching inside data streams. Data delivered as streams requires *continuous queries (CQs)*, which are queries over streams that continuously deliver new results as new data arrives to the queried stream. For example, if some machines continuously deliver streams of temperature readings, a continuous query may be:

> ‘Continuously show me the temperature readings for sensor X on equipment of
> model Y operating at 20% higher temperature than what is recommended.’

2.6 Data Stream Management System (DSMS)

CQs are sent to a data stream management system (DSMS) for execution. Once a continuous query is started to be executed it will deliver results until some *stop condition* is fulfilled, e.g. at some particular time point or when the user signals that the continuous query result delivery should stop.

In the Smart Vortex project CQs are used for instance to monitor critical parameters of products-in-use, to monitor how operators use the equipment, and to support collaborative interactions.

2.7 Linear Road Benchmark (LRB)

The *Linear Road Benchmark (LRB)* [2] is a benchmark for measuring the performance of a DSMS. LRB consists of a data generator that during a three hour simulation generates streams of events simulating traffic on expressways where every single vehicle is monitored. The load (number of events) is continuously increased during the simulation. Continuous queries dynamically identify accidents and signal toll alert events. The performance of a DSMS is measured by how many expressways it can handle, called its *L-rating*.

2.8 Meta-data

Meta-data, often referred to as "data about data" can be defined as "...structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource." This can for instance be information about the origin of the data (the data source), the purpose of the data, the time and date of creation, etc.

The content of a regular DBMS is represented as a meta-database called the *database schema*.

3 DATA STREAM MANAGEMENT SYSTEMS

In this section some well known DSMSs implementing various query languages for data streams are overviewed.

3.1 Stanford STREAM

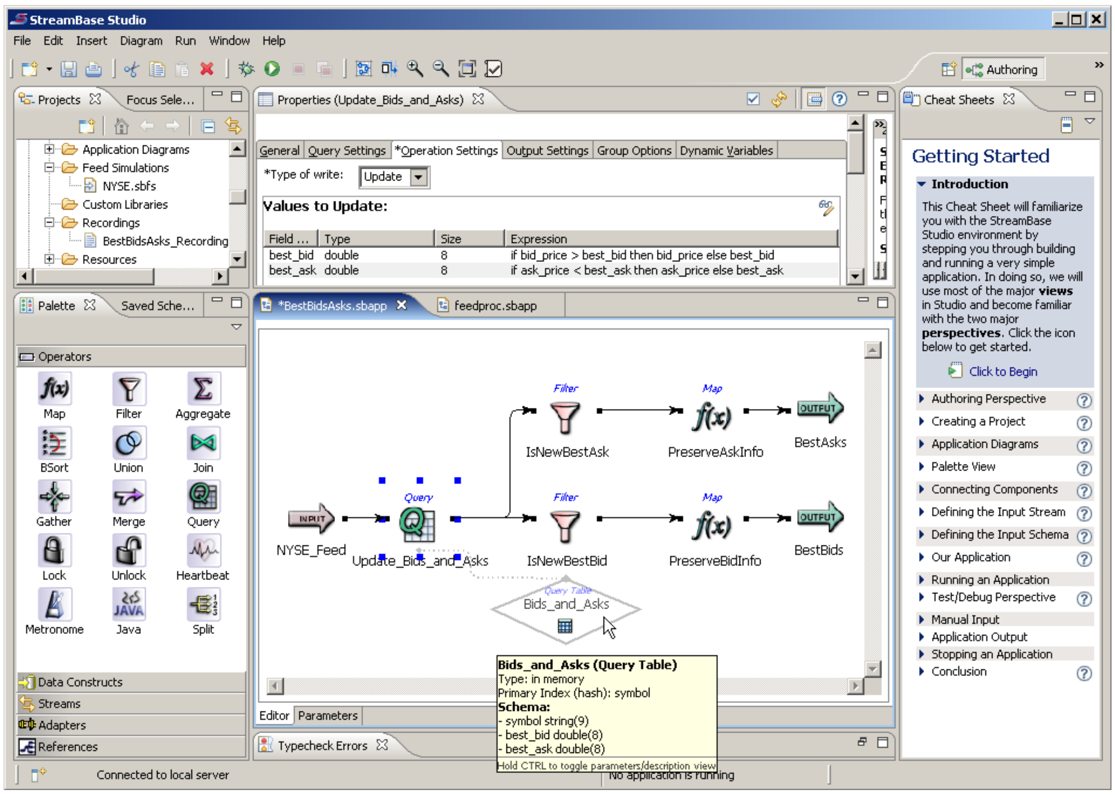
The Stanford Stream Data Manager (STREAM) [12] was a project at Stanford university with the goal of developing a DSMS capable of handling large volumes of queries in the presence of multiple, high volume, input streams and stored relations. The project produced a DSMS prototype and created CQL, a declarative query language based on SQL for expressing continuous queries on streams. The fully functional prototype is available for download on the STREAM homepage.

The Linear Road Benchmark has been defined using CQL and STREAM but we are not aware of any L-rating for STREAM.

3.2 StreamBase

StreamBase [14] 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. The StreamBase DSMS claims to offer “the fastest performance, with the lowest latency and highest throughput" [15]. 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 1. While the languages are conceptually very different they are equal in their expressiveness.



*Type of write: Update

Values to Update:

Field ...	Type	Size	Expression
best_bid	double	8	if bid_price > best_bid then bid_price else best_bid
best_ask	double	8	if ask_price < best_ask then ask_price else best_ask

Bids_and_Asks (Query Table)
Type: in memory
Primary Index (hash): symbol
Schema:
- symbol string(9)
- best_bid double(8)
- best_ask double(8)

Hold CTRL to toggle parameters/description view

Getting Started

Introduction

This Cheat Sheet will familiarize you with the StreamBase Studio environment by stepping you through building and running a very simple application. In doing so, we will use most of the major **views** in Studio and become familiar with the two major **perspectives**. Click the icon below to get started.

Click to Begin

- Authoring Perspective
- Creating a Project
- Application Diagrams
- Palette View
- Connecting Components
- Defining the Input Stream
- Defining the Input Schema
- Our Application
- Running an Application
- Test/Debug Perspective
- Manual Input
- Application Output
- Stopping an Application
- Conclusion

Figure 1: StreamBase Studio built upon the popular ECLIPES IDE

The Aurora prototype was in 2004 measured to provide a LRB L-rating of L=2.5 on a single processor. This means that the DSMS has the capacity to handle the traffic intensity generated for 2.5 motorways. (i.e. both directions of two motorways and one lane of a third motorway). We are not aware of any L-rating for the StreamBase product.

3.3 SCSQ

Super Computer Stream Query processor, SCSQ, [22][23][24] is a DSMS prototype developed at Uppsala University where the CQs are specified in a query language called SCSQL that includes types and operators for sets, streams, and vectors. Vector processing operators enable queries to contain numerical computations over the input data streams. Composite types are allowed, which enables useful constructs such as vector of stream. The system is extensible so that programmer can define customized data structures and query operators. SCSQ has been implemented to execute in a variety of hardware environments, including desktop PCs, Linux clusters, and IBM BlueGene.

When executing an expensive CQ over streams of high rate, it is important that the CQ keeps up with the rate of the input stream. One strategy to keep up with the stream rate in overload situations is *load shedding* [20]. This is not an option if data loss is not tolerated. If the input stream is bursty, it may be feasible to balance the load over time by writing some tuples to disk during overload, and process them later during quieter periods. If the input stream rate is constantly high and if the application needs the DSMS to respond in time, state spill is not an option. One approach to keep up with the input stream is to parallelize the execution, which is the approach used in SCSQ.

By using data parallelism for LRB it has been shown in [22] that SCSQ can make continuous query processing communication bound and not limited by the processing speed even for expensive computations. Thus with SCSQ the capacity (L-rating) depends on the wire speed only - not the processor speed.. Previous published LRB DSMS measurements were always compute bound, i.e. limited by processor speeds. This result enables expensive computations and decision support algorithms to be directly applied on streaming data in real-time. As a proof of concept, we implement LRB in SCSQ, and achieve two orders of magnitude higher L-rating than other implementations [21].

The superior continuous query processing rate of SCSQ is enabled by the combination of two key technologies: parallelization of stream splitting in conjunction with the use of physical windows. Parallelizing stream splitting speeds up the distribution of data in a federated DSMS, whereas the use of physical windows saves communication cost.

3.4 Wavescope

WaveScope is an MIT research project in the field of high data-rate wireless sensor networks [21]. One of its contributions, most relevant to this report, is the development of a declarative functional language for stream and signal processing called WaveScript. No L-ratings are reported.

3.5 System S

IBM’s System S [19] was a DSMS developed by IBM Research for high processing of high volume events for different application. The first parallel implementation of LRB was made on System S achieving an L-rating of 2.5 [8]. System S did not have a particular query language but instead the programmer can define operators through a conventional programming language that can be combined into data flows by a scripting language.

3.6 Coral 8

Coral8 now owned by SAP is a DSMS for processing complex events. Its main application is financial servicing, which requires stream processing with very low latency. Relatively simple filters and decision support algorithms are applied on the processed streams. Coral8 has an SQL-based continuous query language called CCL, which has similar functionality as CQL. The system can be configured for parallel processing of high volume events. We have not been able to find any detailed documentation or L-rating of Coral8.

3.7 Stream Insight

Microsoft’s StreamInsight [17] is a temporal main-memory query processing engine over complex event streams integrated in the .NET framework. Program modules can be defined that reads streams of events and iteratively processes the events to produce new (derived) streams of events. Each event has an associated life span (time interval) and the processing is often based on transforming and aggregating such time oriented events. A library of window forming primitives transform the streams into windows of events over which queries can be stated using the procedural query language LINQ [11]. The system is extensible by allowing the programmer to define own operators and aggregate functions in C#. No L-ratings are reported.

4 CONTINUOUS QUERY LANGUAGES

DSMS queries are different from conventional database queries in, for instance 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 (often called events). Furthermore, stream queries are 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. The semantics of different CQ operators was recently investigated in [10].

In this section, we first take a closer look at the continuous query language CQL to illustrate the concepts. Then other stream processing query languages are investigated.

4.1 CQL

CQL is syntactically very similar to the SELECT statement of SQL making it easy to learn and understand for users who have had 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]. The operators include many of those normally found in SQL, such as projection, selection, aggregation, joining, grouping, etc. This approach was chosen so that well understood relational concepts could be reused and extended.

Beyond SQL, 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 of SQL they offer 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.

4.1.1 Stream-to-relation operators

The stream-to-relation operators in CQL are based on *sliding windows* [4], which can be thought of as a continuously changing 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*, *tuple based*, and *partitioned windows*. In the following paragraphs, each will be described through a series of examples.

4.1.1.1 Time based windows

Given the following streams:

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

Each auction tuple (event) 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 final 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 ";
```

The query uses the notation *[RANGE 1 DAY]* to define a *time based sliding window* on the *Purchases* stream, which produces a 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.

4.1.1.2 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 ";
```

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.

4.1.1.3 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 ;
```

The *partitioned window* operator expects a stream *S* and a positive number of tuple attributes in the window as 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* (the query output) is formed by taking the union of all sub-relations.

4.1.2 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 “Return a stream of sellers of the most expensive item sold, 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 );
```

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*.

4.1.3 Processing tuples

A fundamental property of CQL is that windows are calculated conceptually using a time-driven model [11]. 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 can be contrasted to a tuple-based model, where a window’s content is re-evaluated every time a new tuple arrives.

The differences between the two models are illustrated with an example. Assume the following stream of purchases:

```
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)
```

The following query is stated:

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

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)
```

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)
```

Further examples and an investigation of the strengths and limitations of both models can be found in [9].

4.1.4 Defaults

Two of the main goals when designing CQL were that simple queries should be easy to write and do what you expect. 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.

4.1.5 Implementation in STREAM

The current implementation of CQL in STREAM does not provide all features described in [3]. Some of the missing features can, however, be expressed if intermediate named queries are used where 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.
2. The HAVING clause is not supported.
3. Arithmetic with aggregations is not supported 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. The INTERSECT operator is not supported, but UNION and EXCEPT are.

The STREAM project has not published an implementation of the LR benchmark. However, they have defined a benchmark specification using CQL [13].

4.2 StreamSQL

StreamSQL [14] uses a similar reasoning as CQL to take advantage of SQL's ubiquity in DBMSs. Thus, StreamSQL shares many syntactic and semantic similarities with both SQL and CQL. The language has operators that 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 from CQL is the addition of operators that deal directly with streams.

StreamSQL uses a tuple-driven model ordering tuples not only on their timestamps, but also on their order of arrival [4]. 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*. This may cause the result to non-deterministic.

A stream can be joined with static tables, combining each stream tuple with a set of tuples from the tables. It is possible to combine two streams using a *UNION* operator, the result being an interlacing of the two streams with their order of arrival preserved.

StreamSQL has pattern matching operators, which allow detection of temporal or range-based patterns between tuples from one or more streams. The *PATTERN* clause consists of a template that references one or more streams. It defines some relationship between the streams 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. either constraining maximum time or maximum range of values on some field during which the relationship must occur. A formal definition of the pattern matching language, together with additional examples, can be found in [16]. The following is an example of a continuous pattern query:

Like CQL, StreamSQL makes use of the notion of sliding windows. Compared with CQL, the StreamSQL window offers a few additional features such as:

1. An offset that species how long to wait before opening the first window on the stream.
2. A timeout after which the window closes.
3. The ability to define windows based 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.

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. For example, the “most expensive item" query above is expressed this way 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 ;
```

Thus 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*.

Compared with StreamSQL, CQL is relatively small in terms of available features. StreamSQL has all the features available in CQL. Furthermore it also has a completely new class of stream-to-stream operators which allow manipulation of streams without intermediate conversion to and from relations. These operators enable stream filtering, joining, pattern matching, etc. However, StreamSQL uses a tuple-driven model and therefore some queries expressible in CQL are not possible when using StreamSQL and vice versa. Research has recently been undertaken in unifying the tuple and time-based models [9], so that the benefits of both can be harvested and their individual limitations avoided.

4.3 SCSQL

The SCSQL query language is a continuous query language implemented in SCSQ extending the functional query language AmosQL [6]. In SCSQL, a *stream* is an object that represents ordered (possibly unbounded) sequences of objects, a *bag* represents relations, and a *vector* represents bounded sequences of objects. For example, vectors are used to represent stream windows, and vectors of streams are used to represent ordered collections of streams.

An important property of SQSQL is that it is extensible so that programmers easily can define own *foreign functions* in some conventional programming language such as C or Java. For example, new kinds of stream event formats produced by particular kinds of communication protocols can be accessed through foreign functions and used in continuous queries. Data filtering or mining algorithms can be defined as foreign functions and used in queries.

Another central feature of SCSQL is the facilities to define distributed and massively parallel queries. The query language includes *stream processes* (SPs) and *parallelization functions*, which allow the user to specify customized parallelization and distribution of queries. This is enabled by high-level primitives for scalable stream splitting and for specifying distributed and parallel computations [22][23]. SCSQ supports data parallelism by a highly parallel stream splitting operator called *parasplit*. A fundamental problem of continuous query plan parallelization is the fact that heavyweight stream operators are bottlenecks. Parallelizing a data stream requires the input stream to be split into parallel sub-streams over which expensive continuous query operators are executed in parallel. In SCSQ the highly scalable *parasplit* operator partitions a stream of high volume into parallel sub-streams. The parallelization primitives enable massive streamed scale-out of continuous queries involving expensive computations. In this way costly decision making algorithms can be applied in real-time over steams.

SCSQ is based on the mediator database system Amos II [6], which includes a main-memory object store and allows different kinds of distributed data sources to be queried. SCSQL extends AmosQL with stream and parallelization primitives.

In summary, SCSQL is an extensible query language for both stored and streamed data. SCSQL allows continuous and ad hoc queries over these data sources to produce derived streams. The foreign function interface of SCSQ allows any external data and processing to be plugged into SCSQ. Finally, the parallelization functions of SCSQL allow stream processing to be massively parallelized.

4.4 Wavescript

Sensors, such as microphones and cameras, which produce signals with data rates of hundreds of thousands of samples per second, often require analyzing the signal in chunks instead of splitting them into a sequence of tuples as in conventional DSMSs. Furthermore, the analyses of such signals is often application specific and must be processed by user defined code which introduces the non negligible overhead of converting the signal data back and forth from the DSMS to some external language such as C or MATLAB for processing.

WaveScript [5] is a programming language that integrates high data rate stream processing with signal processing. WaveScript groups signal samples into chunks, called *signal segments*. Each signal segment contains a fixed number of signal samples emitted at a regular rate, allowing the segment to timestamp only the first sample while the remaining samples are implicitly timestamped. By doing so the system avoids 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., or users may define new operators directly using WaveScript, thus avoiding data conversion to and from foreign functions defined in an external language.

Signal segments are similar to physical windows in SCSQ. In both SCSQ and WaveScript, the idea is to save communication and processing overhead by grouping tuples together in chunks.

4.5 StreamInsight queries

StreamInsight uses the LINQ programming language [17][11] to express queries. The programming style in LINQ is not declarative and set oriented as in query languages such SQL and CQL, but rather the ‘queries’ are expressed as FOR-loops defining nested iterations over nested collections while assigning local variables. StreamInstight provide such LINQ based iteration over streams. Different kinds of windows over streams can be defined and iterated over. A library of window forming primitives is used for successively generating streams of the different kinds of windows. Over each so streamed window it is possible to define filters and transformations using the LINQ programming language. Streams can be joined window by window using nested iterators. Streams can be split using a multicast operator. The output from a LINQ program is another stream, i.e. a derived stream in Smart Vortex vocabulary. The semantics of StreamInsight with LINQ is very much oriented towards complex processing of events with associated ‘payload’ values lasting over intervals of time.

4.6 Coral8 queries

The query language of Coral8 is called CCL [18] and is similar to CQL. The queries are expressed in an SQL-like syntax over windows of events. Foreign functions can be defined for doing computations in streams. SQL back-end databases can be joined in CQs.

4.7 Streaming SPARQL

The most well known implementation of extending SPARQL for continuous queries over streams is [5] implemented on top of the Stanford STREAM system. Window and aggregation primitives are there added to the SPARQL language. One problem is that while SPARQL is based on sets of triples the semantics of streaming SPARQL is not clear. Essentially the elements of the streams should be assigned an order, for example by timestamps. This is the approach in [5] where the time stamp of an object is the time when it last arrived to the DSMS in some triple. An alternative is that the time stamp is associated with the triples themselves rather than URIs. This requires further investigations.

5 CONCLUSIONS

The Smart Vortex project will require customized continuous query processing over different kinds of streams originating from industrial contexts. Furthermore, data originating in different kinds of streams need to be combined with data in regular databases. It is therefore important to design an extensible federated DSMS where new kinds of both streaming and regular data sources can be incorporated in queries.

Some of the processing to be made over streaming data will require more or less advanced computations (e.g. statistical) in real-time. It is therefore important to provide easy-to-use and flexible mechanisms to extend the system with application dependent functions called in continuous queries. The federated DSMS must be extensible with customizable computations.

Some of the computations made in real-time may be relatively expensive to compute. For example, there is need to compute vibration frequencies in real-time over data streams in order to detect resonances. Therefore the federated DSMS must have the ability to perform more-or-less expensive computations for decision support in real-time over streams. Here, real-time means that the system must on the average be able to keep up with the stream flow while making the necessary computations.

The conclusion is that the most suitable query language and system for building a federated DSMS to meet the requirements posed by the Smart Vortex project is SCSQ. The federated DSMS should be built by extending SCSQ and SCSQL with new facilities.

6 REFERENCES

[1] D.J.Abadi, D.Carney, U.Cetintemel, M.Cherniack, C.Convey, S.Lee, M.Stonebraker, N.Tatbul, and S.Zdonik: Aurora: a new model and architecture for data stream management, *The VLDB Journal*, 12(2):120-139, 2003.

[2] A.Arasu, M.Cherniack, E.Galvez, D.Maier, A.S.Maskey, E.Ryvkina, M.Stonebraker, and R.Tibbetts: Linear road: a stream data management benchmark. *Proceedings of the 30th international conference on Very large data bases (VLDB 2004)*, pages 480-491, 2004.

[3] A.Arasu, S.Babu, and J.Widom: The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121-142, 2006.

[4] B.Babcock, S.Babu, M.Datar, R.Motwani, and J.Widom: Models and issues in data stream systems. *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS2002)*, pages 1-16, New York, NY, USA, 2002.

[5] D.F.Barbieri, D.Braga, S.Ceri, M.Grossniklaus: An Execution Environment for C-SPARQL Queries, *Proc. EDBT Conf.*, 2010.

[6] S.Flodin, M.Hansson, V.Josifovski, T.Katchaounov, T.Risch, M.Sköld, and E.Zeitler: *Amos II Release 13 User's Manual*, http://www.it.uu.se/research/group/udbl/amos/doc/amos_users_guide.html, 2011.

[7] L.Girod, Y.Mei, R.Newton, S.Rost, A.Thiagarajan, H.Balakrishnan, and S.Madden: The Case for a Signal-Oriented Data Stream Management System. In *CIDR 2007*, pages 397-406, www.crdrdb.org, 2007.

[8] N.Jain, et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core, *Proc. SIGMOD Conf.*, 2006.

[9] N.Jain, S.Mishra, A.Srinivasan, J.Gehrke, J.Widom, H.Balakrishnan, U.Cetintemel, M.Cherniack, R.Tibbetts, and S.Zdonik. Towards a streaming SQL standard. *Proc. VLDB Endowment*, 1(2):1379-1390, 2008

[10] Y-N.Law, H.Wang, C.Zaniolo: Relational languages and data models for continuous queries on sequences and data streams, *ACM Transactions on Database Systems (TODS)*, 36(2), May 2011

[11] LINQ, http://msdn.microsoft.com/en-us/library/bb308961.aspx.

[12] Stanford Stream Data Manager, http://infolab.stanford.edu/stream/.

[13] Stanford. STREAM Linear Road Benchmark Specication. http://infolab.stanford.edu/stream/cql-benchmark.html .

[14] StreamBase home page, http://www.streambase.com/.

[15] StreamBase: "Why StreamBase", http://www.streambase.com/solutions-streambase.htm.

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

[17] Stream Insight Team: A Hitchhiker's Guide to StreamInsight Queries, http://blogs.msdn.com/b/streaminsight/archive/2010/06/08/hitchhiker-s-guide-to-streaminsight-queries.aspx

[18] Sybase, Inc.: Stream, Schema, Adapter, and Parameter CCL Language Extensions, http://www.sybase.com/files/Technical_Documents/SY_Coral8_StreamSchemaAdapterandParameterLanguage_TechDoc.pdf.

[19] System S – Stream Computing at IBM Research, http://public.dhe.ibm.com/software/data/sw-library/ii/whitepaper/SystemS_2008-1001.pdf.

[20] N. Tatbul et al: Load shedding in a data stream manager. *Proc. VLDB*, 2003.

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

[22] E.Zeitler and T.Risch: Massive scale-out of expensive continuous queries, presented at *37th International Conference on Very Large Databases, VLDB 2011*, in *Proceedings of the VLDB Endowment*, Vol. 4, No. 11, 2011, .

[23] E.Zeitler: *SCSQ user's guide:*, UDBL, Dept. Of Information Technology, Uppsala University, 11 Aug. 2011, http://www.it.uu.se/research/group/udbl/publ/scsql.pdf.

[24] E.Zeitler: *Scalable Parallelization of Expensive Continuous Queries over Massive Data Streams*, Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 836 ISSN 1651-6214, ISBN 978-91-554-8095-0, Acta Universitatis Upsaliensis, 2011, http://www.it.uu.se/research/group/udbl/Theses/ErikZeitlerPhD.pdf.