Large Scale Integrating Project

Grant Agreement no.: 257899

D3.3: Specification of query language extensions

SMART VORTEX –WP3-D3.3

Project Number	FP7-ICT-257899
Due Date	31.03.2012
Actual Date	24.03.2012
Document Author/s:	Tore Risch, Erik Zeitler, John Lindström, Mathias Johansson
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)**
0.1	120423	First version	TR, EZ
0.2	120424	Revised version	TR, EZ, JL, MJ

Document Change Commentator or Author		
Author Initials	**Name of Author**	**Institution**
TR	Tore Risch	Uppsala University
EZ	Erik Zeitler	Uppsala University
JL	John Lindström	Luleå University of Technology
MJ	Mathias Johansson	Alkit Communications

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
0.2	26.09.2012	Minor changes	IK

Catalogue Entry

Title	Specification of query language extensions
Creators	
Subject	
Description	
Publisher	
Contributor	
Date	
ISBN	
Type	
Format	
Language	English
Rights	

Citation Guidelines

EXECUTIVE SUMMARY

This document specifies the data stream query language (DSQL) chosen for the 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 to design an extensible federated Data Stream Management System (FDSMS) where both streaming and regular data sources can be incorporated and where continuous queries (CQs) are expressed using a DSQL where queries over federations of different kinds of distributed data streams can be expressed.

Parts of the data processing will require advanced computations (e.g. statistical) made in real-time over streaming data. To cope with this challenge, the DSQL must be extensible to allow application dependent functions to be called in CQs. Some of the computations made in real-time may be relatively expensive to compute. Therefore the DSQL must have facilities to perform expensive computations for decision support in real-time over streams.

The conclusion is that, based on the above requirements, novel continuous query language facilities are needed for building the FDSMS of the Smart Vortex project. The existing prototype implementation of the data stream query language SCSQL will be extended with required functionality.

TABLE OF CONTENTS

1. INTRODUCTION

A central technology in Smart Vortex is the ability to search and analyze high volume data streams in distributed environments with means of a data stream management system (DSMS). The search and analysis are specified using **continuous queries** (CQs) expressed in a **data stream query language** (DSQL).

It should be noted that there is not yet any standard proposal for a DSQL. Many of the principles for how to define such a query language are still being debated within the scientific community and literature [4][5].

A 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 real-time data streaming from one or multiple sources. In Smart Vortex many different kinds of data stream sources originating from different kinds of equipment need to be processed, which leads to the need for a ***federated DSMS*** *(FDSMS)*.

Smart Vortex has some special requirements, not met by the existing DSQLs investigated in D3.2. In particular there is a need for a DSQL where combinations of streams from different kinds of data streams originating from different kinds of equipment can be queried. Different kinds of numerical algorithms will be applied on the extracted data streams for data analysis, reduction, and transformation. The DSQL must therefore be *extensible* so that customized operators can be defined.

The data streams to be processed by the Smart Vortex FDSMS are furthermore highly distributed. Data stream processing can take place directly in the equipment, at the site where the equipment is located, at some central cluster computer at an engineering site, or even in an analyzing engineer's workstation. It must therefore be possible to specify *distributed* CQs where different pieces of the computations required are executed at different distributed sites.

All this leads to the requirement to base the query language used in Smart Vortex on an ***extensible DSQL*** where ***distributed CQs*** can be expressed. The project will require extensions of DSQL technology in order to meet additional performance and functionality requirements by the Smart Vortex applications.

It is a great advantage to base the query language on technology where we have prototype implementations that fulfill the functional requirements and which can be extended for new requirements by Smart Vortex. In the UDBL group we have developed such a prototype system called **Super Computer Stream Query Processor** (SCSQ) with a **query language SCSQL**. We have chosen SCSQL as the query language to use in the FDSMS for the Smart Vortex project, as it is the state-of-art and we are in control of its implementation.

2. SCALABLE AND EXTENSIBLE DATA STREAM QUERY LANGUAGE

Super Computer Stream Query processor, SCSQ, [8] [9] is a DSMS prototype developed at Uppsala University, where the CQs are specified in a query language called SCSQL which 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 the 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* [7]. 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 allowing data parallelism to be specified in the DSQL it has been shown in [8] that SCSQ can make continuous query processing *communication bound* and not limited by the processing speed, even for expensive computations. Thus with SCSQL the capacity depends on the wire speed only - not the processor speed. This result enables expensive computations and decision support algorithms to be directly applied on streaming data in real-time, which is required to do intelligent decisions based on high volume events streaming from industrial equipment in Smart Vortex.

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 [8].

The SCSQL query language is a DSQL implemented in SCSQ extending the functional query language AmosQL [1]. In SCSQL, a *stream* is an object that represents ordered (possibly unbounded) sequences of objects, a *bag* represents regular database 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 Java, Python, or C. 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 facility to define distributed and massively parallel queries. The query language includes *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 [8]. 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. Parasplit enables 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 [1], which includes a main-memory object store and allows different kinds of distributed data sources to be queried. Each SCSQ node thus includes its own main-memory database where meta-data can be stored and queried. Typically CQs match combinations of incoming streaming data with meta-data in the local database. The data model used for the local database is semantic in the sense that complex data relationships can be defined declaratively using a functional and object-oriented data model. A local database may also contain wrappers of external databases, e.g. to retrieve meta-data from a regular relational database using SQL.

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.

3. THE SCSQL QUERY LANGUAGE

This section presents the important new features in the data stream query language SCSQL. It extends the query language AmosQL [2] with stream and parallelization primitives. Please refer to [2] for those query language constructs SCQL have in common with AmosQL.

The APIs for Java is documented in [1]. This API currently exactly the same as the APIs for Amos II. The differences from the programmer's point of view is the ability to handle indefinite scans over streams, rather than finite conventional scans over query results. The corresponding API for C is documented in [6].

In this section only some of the parallelization primitives of SCSQL are described. The query language primitives of AmosQL [2] are supported also by SCSQL. Important is that SCSQL is an extensible system so new SCSQL functions can be defined as new requirements are introduced by Smart Vortex. Refer to [8] and [3] for examples of more advanced usage of SCSQL.

3.1 Basic SCSQ stream execution

A single SCSQ server is started by calling `scsq` (`scsq.exe` in Linux) at the command prompt. In order to enable several SCSQ servers, a *SCSQ coordinator* must be running. The script `swc` (`swc.sh` in Linux) starts SCSQ and a SCSQ coordinator in the background. The SCSQ coordinator can then start new SCSQ servers on any node in a cluster.

All data in SCSQ is represented by *objects* in SCSQL. The datatype *stream* represents possibly unbounded sequences of any kind of objects. For example, the result of a continuous sub-query is a stream.

Parallel and distributed stream computations can be defined in SCSQL using *parallelization functions*.

3.2 Parallelization functions

A *parallelization function* operates on collections of streams, and is used for specifying parallel executions queries over streams. Figure 3.1 illustrates three basic classes of parallelization functions; *splitstream*, *mapstream*, and *mergestream* functions.

A *splitstream function* splits an input stream into two or more output streams. The number of output streams of a splitstream function is called its *width*. A *mapstream function* applies a stream function (i.e. a query) on each stream in a collection of streams. Finally, a *mergestream function* merges (or joins) a collection of streams into a single output stream. Examples of stream merge functions implemented in SCSQ are: *ustreams()*, *zipstreams()*, and sort-merge-join of vectors, *mergestreams()*. All other stream merging functions are variants of these three basic functions.

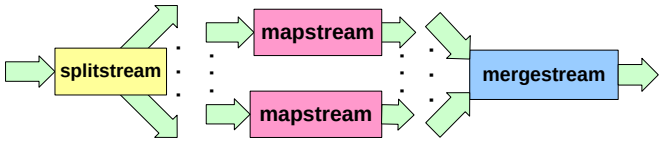

Figure 3.1. Parallelization functions.

3.2.1 Splitstream functions

A splitstream function has the signature:

```
splitstream(Stream s, Integer w, Function rfn, Function bfn)
-> Vector Of Stream sv
```

The input stream *s* is split into *w* output streams in the vector *sv*. The first functional argument *rfn* is the *routing function*, having signature

```
rfn(Object tpl, Integer w) -> Integer,
```

which returns the output stream number (between 0 and *w* – 1) for each tuple that should be routed to a single output stream. The *broadcast function* with signature `bfn(Object tpl) -> Boolean`

returns `true` for tuples to be broadcasted to all output streams. *bfn* and *rfn* return `nil` or `false` for tuples that should neither be broadcasted nor routed. *rfn* and *bfn* are defined declaratively in the query language by the user.

The following example shows how to split a stream of integers into two streams of integers, one containing odd numbers and the other one containing even numbers:

splitstream(siota(1, 100000000), 2, #'modq', #'f');

The function *modq()* is defined as:

```
create function modq(integer i, integer q) -> integer as
mod(i, q);
```

When used as a routing function, *modq()* will route integers to output stream number *i* % *q*, where *q* is the number of output streams. To define an empty broadcast function we use the *false* function:

```
f(Object o) -> Boolean
```

which returns `false` for all `o`. Using *f()* as a broadcast function, no tuples will be broadcasted.

3.2.2 Mapstream functions

The function *mapstreams()* has the signature:

```
mapstreams(Vector of Stream sv, Function mapfn) -> Vector of
Stream sw
```

mapstreams() applies the function (query) *mapfn()* over each stream in the vector of streams *sv*. Each element sw_i of the ouput vector of streams *sw* contains the result of applying *mapfn()* over sv_i. The *mapfn()* used in *mapstreams()* must have the signature

```
mapfn(Stream s) -> Stream r
```

3.2.3 Merging streams

The function *ustreams()* (union-all) merges the tuples of its input stream vector. The signature of *ustreams()* is:

```
ustreams(Vector of Stream vs) -> Stream
```

The vector of stream *vs* contains one or more streams. Whenever a tuple is available on any of the streams in *vs*, *ustreams* moves that tuple and emits it to the output stream.

For example, the following query makes a union-all of tuples from a broadcast:

```
ustreams(splitstream(siota(1, 10000000), 2, #'f', #'t'));
```

The function *t(Object o) -> Boolean* returns true for all o. Using *t()* as *bfn* and *f()* as *rfn* turns *splitstream()* into a broadcaster.

The query returns a new stream as result without running it. To retrieve the tuples (events) from the stream use:

```
in(ustreams(splitstream(siota(1, 100), 2, #'f', #'t')));
```

3.2.4 Pairing streams

The function *zipstreams()* pairs tuples from a vector of streams by their order of arrival. The signature is

```
zipstreams(Vector of Stream vs)->Stream of Vector
```

As soon as each input stream `vs[i]` , $i = 0 \ldots d-1$ has a new tuple t_i available, `zipstreams(vs)` moves the tuple from each input stream and emits a vector of tuples $\{t_0, t_1, \ldots t_{d-1}\}$ on its output stream. The dimensionality of the output stream vector is the same as the number of input streams.

For example, the following query returns pairs of consecutive odd/even numbers:

```
in(zipstreams(splitstream(siota(1, 100), 2,
              #'modq', #'f')));
```

3.2.5 Matching streams

The function *mergestreams()* matches tuples arriving from a vector of streams based on some value (e.g. time stamp or sequence number) into a single stream. The signature is

```
mergestreams(Vector of Stream of Vector vs, Integer attrib) ->
Stream of Vector
```

As soon as each input stream *vs[i]* , $i = 0 \ldots d-1$ has a new tuple w_i available, *mergestreams(vs)* moves all tuples w_i that have the smallest attribute value to the output stream. When all such tuples are emitted, *mergestreams()* waits until all streams have a new tuple w_i and repeats.

For example, this function computes two streams of numeric vectors in parallel and delays their emit frequencies (using *retard()*) before they are merged in the order of position 0 of the vectors.

```
in(mergestreams(mapstreams({
  streamof(retard(0.5, {iota(10,15), 1})),
  streamof(retard(0.5, {2*(iota(5,8)), 2}))
},#'id'), 0));
```

4 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 FDSMS 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 FDSMS 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 FDSMS 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 DSQL to meet the requirements posed by the Smart Vortex project is the AmosQL with its streaming extensions in SCSQL. The federated DSMS should be built by extending SCSQ and SCSQL with new facilities required by Smart Vortex.

REFERENCES

[1] D.Elin and T.Risch: *Amos II Java Interfaces*, UDBL, Dept. Of Information Technology, Uppsala University, http://user.it.uu.se/~torer/publ/javaapi.pdf, 2000.

[2] S.Flodin, M.Hansson, V.Josifovski, T.Katchaounov, T.Risch, M.Sköld, and E.Zeitler: *Amos II Release 14 User's Manual*, http://www.it.uu.se/research/group/udbl/amos/doc/amos_users_guide.html, 2012.

[3] G.Gidofalvi, T.B. Pedersen, T.Risch, and E.Zeitler: Highly Scalable Trip Grouping for Large Scale Collective Transportation Systems, *Proc. 11th International Conference on Extending Database Technology, EDBT 2008* , Nantes, France, March 2008.

[4] 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

[5] 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

[6] T.Risch: *Amos II C Interfaces*, UDBL, Dept. Of Information Technology, Uppsala University, http://user.it.uu.se/~torer/publ/externalC.pdf, 2012.

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

[8] 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, .

[9] 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.