IT 10 059

Examensarbete 30 hp

Oktober 2010

Processing Functional CQL Queries

Robert Kajic

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

Processing Functional CQL Queries

Robert Kajic

At UDBL (Uppsala DataBase Laboratory) we are developing the DSMS SCSQ (SuperComputer Stream Query processor) based on the main memory DBMS Amos II. Amos II is a functional DBMS where data and information are represented as typed functions. In SCSQ database queries over streams are expressed in the functional query language SCSQL, a language similar to the object oriented parts of SQL:99 but extended with parallel stream query facilities.

In this paper we investigate what existing functionality in SCSQ and Amos II can be utilized to support CQL, a continuous query language developed by the Stanford STREAM project. SCSQL is extended with functionality required to support CQL. The extended functional stream query language is called FCQL. To implement FCQL, SCSQ is extended with new operators that adhere to the semantics of CQL. FCQL is a functional continuous query language with the same expressive power as CQL. Furthermore, we show how CQL queries can be translated to FCQL in a systematic way and by doing so give a template for an automatic CQL-to-FCQL translator. We also evaluate the completeness of FCQL by translating to FCQL the queries of the linear road DSMS benchmark as it was expressed in CQL by the Stanford STREAM project.

Handledare: Tore Risch
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 10 059
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 a DBMS allows searching only stored data, while a DSMS in addition provides query facilities to search directly in data streaming from some source. 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* in that they run until they are terminated, while conventional queries are executed on demand and run until all requested data is delivered.

At UDBL (Uppsala DataBase Laboratory) we are developing the DSMS SCSQ (SuperComputer Stream Query processor) [31, 32, 28] based on the main memory DBMS Amos II [15, 16, 24, 25]. Amos II is a functional DBMS where data and information are represented as typed functions. In SCSQ database queries over streams are expressed in SCSQL [31], a query language similar to the object oriented parts of SQL:99 but extended with parallel stream query facilities. SCSQL is an extension with stream and parallelization primitives of the Amos II query language AmosQL [27].

There are several query languages developed for DSMSs, CQL (Stanford STREAM) [1, 2, 8], StreamSQL (StreamBase) [21], WaveScript (MIT) [29], and SCSQL [28]. This project aims to provide CQL support integrated with SCSQ.

In a previous project [17], we investigated 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. An overview and comparison of SQL, CQL, StreamSQL and WaveScript was also given.

In this project we investigate what existing functionality in SCSQ and Amos II can be utilized to support CQL. This investigation is to a large extend based on the results from [17]. In cases where Amos II and SCSQ are lacking in functionality required to support CQL, SCSQ is extended with new operators which adhere to the semantics of corresponding STREAM CQL operators. The result is FCQL, a functional continuous query language with the same expressive power as CQL. Furthermore, we will show how CQL queries can be translated to FCQL in a systematic way and by doing so give a template for an automatic CQL-to-FCQL translator. We also evaluate the completeness of FCQL by translating the linear road benchmark as it was expressed in CQL by the Stanford STREAM project [19].

2 Background

2.1 Data Stream Management Systems

Data stream management systems are presented with the problem of processing continuous, often high-volume, and infinite streams of data. Such data streams can not be processed using traditional relational operators, which are defined for finite relations. A common approach taken by several DSMS systems, in processing data streams, is to form *sliding windows* upon the data stream [5, 2, 4]. The sliding window can be thought of as view upon a stream that, at any point in time, reflects the viewed part of the stream as a finite relation. As time flows and new tuples arrive, the window moves over the stream and the content of the relation is changed to reflect the current view. This continuously changing view is called a *continuous relation*, and by being finite it can be processed by relational operators. Continuous relations are created by *windowing* operators; those relevant to this project are *time*, *counting*, and *partitioned* windowing operators.

2.1.1 Windowing Operators

All windowing operators are common in that they specify a window *size* and a window *slide*.

For a *time window* size and slide are specified in terms of time units. The sliding window may contain any number of stream tuples as long as the tuples' timestamps (all tuples have an associated *timestamp*) are larger than the window *starting timestamp*, and smaller than or equal to its *ending timestamp* ($startingtimestamp + size$). When the window is full, i.e., a new stream tuple has a timestamp larger than the window ending timestamp, the window increments its starting timestamp by *slide*.

For a *counting window* size and slide are specified as numbers of tuples. That is, the maximum number of tuples that a counting window may contain is *size*. Once full, the window drops *slide* of its oldest tuples.

For a *partitioned window* size and slide are also specified as numbers of tuples, as in the counting window. Unlike counting window, a partition window will split the stream into sub-streams such that each is uniquely identified by one or several stream tuple attributes — much like the *GROUP BY* operator found in the relational algebra. Each sub-stream will then be processed by a counting window using the given size and slide. Finally, the resulting continuous relation is formed by taking the union of all counting windows.

The partitioned window operator is often used to construct continuous relations on which grouping and aggregation can later be meaningfully applied.

2.2 The Stanford Stream Data Manager

The Stanford 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 and high volume, input streams and stored relations [8].

The project produced a DSMS prototype and created CQL [17] — a declarative query language based on an extension of SQL — for expressing continuous queries on streams. The fully functional prototype is available for download on the STREAM homepage [20].

One of the main goals of FCQL is to have the same expressive power as STREAM CQL. That is, all queries expressible in CQL should also be expressible in FCQL.

2.2.1 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 that operate on continuous relations [2, 1, 30]. 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, filtering, aggregation, joining, grouping, etc.

Additionally, CQL has *stream-to-relation* and *relation-to-stream* operators that convert streams to continuous relations and continuous relations to streams. 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.

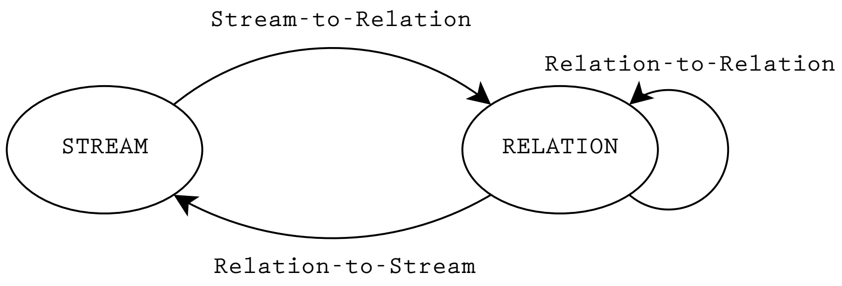

Figure 1: Streams are converted to continuous relations using CQLs sliding window operators. Continuous relations are manipulated using standard relational operators and can be converted back to streams using one of the relation-to-stream operators available in CQL.

2.3 Linear Road Benchmark

Congested roads during rush hours are an ever increasing problem in and around big cities. One method of alleviating this problem is through the use of variable tolling [3]. Tolls are there based on the time of day and/or the current traffic situation in each vehicles vicinity (congestion, accidents, etc.) [7]. 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) [6]. LRB simulates a fictional city with a number L of expressways where tolls are determined through variable tolling. 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.

In this project the LR benchmark is used to evaluate the completeness of FCQL. In appendix A, the complete LR benchmark specification, as defined by the Stanford STREAM project in [19], is translated to FCQL.

2.4 Amos II and AmosQL

Amos II [15, 16, 24, 25] is a distributed mediator system which allows different data sources to be reconciled through a wrapper-mediator approach. In such a system *wrappers* provide access to different data sources through a common data model while the *mediators* provide coherent views of the data provided by the wrappers.

At its core, Amos II is a main memory, object oriented, extensible DBMS with all the facilities normally found in database management systems. Those include a storage manager, a recovery manager, a transaction manager and a functional, declarative and relationally complete query language called AmosQL [15, 16, 27]. AmosQL is based on the functional query languages OSQL [11] and DAPLEX [18]. In AmosQL, queries are specified using the SELECT — FROM — WHERE syntax similar to SQL.

The basic concepts of the data model of Amos II are *objects*, *types* and *functions*. *Objects* are used to represent all entities in the database, both user-defined objects representing real-world entities and system defined objects such as *numbers*, *strings* and *collections* of other objects. All objects are instances of one or more *types* and types are structured in a supertype/-subtype hierarchy. Finally, *functions* describe the properties and semantics of objects and are used to perform calculations on, and to define relations between, objects.

Amos II can store local data in its own internal main memory database and/or provide access to external data sources through its wrappers [26]. Wrappers are available for such varying data sources as Internet search engines [10], music files, CAD systems [13], semantic web metadata [14], other relational databases, other Amos II systems, etc.

All new FCQL operators are defined as AmosQL functions and thus FCQL extends AmosQL with continuous query primitives.

2.4.1 External Interfaces

There are external interfaces between Amos II and the programming languages ANSII C, ALisp, and Java [23]. Through these interfaces Amos II functionality can be extended; external programs can use the *callin* interface to make use of Amos II functionality and furthermore, using the *callout* interface, foreign AmosQL functions can be implemented in C, ALisp or Java. Those foreign functions can then be called by other AmosQL functions. In FCQL, we make use of the *callout* interfaces to C and ALisp to implement our new operators.

C The Amos II external C interface is intended to be used when extensions to the Amos II kernel must be made, or time-critical functionality needs to be implemented [23]. In FCQL, we defined a new datatype, the *stream window*, to represent continuously updated relations [4]. The stream window data type is used in the implementation of our FCQL operators.

ALisp ALisp is an interpreter for a subset of CommonLisp [9] built on top of the storage manager of the Amos II database system [22]. When compared to the external C interface, the ALisp interface is simpler to use

— mainly because memory management is handled automatically by a built-in garbage collector. However, the interface to C is faster and preferred for time-critical code or in cases where external C libraries must be used. Most of the functionality available in Amos II is implemented in ALisp, the same is true for FCQL where ALisp was used to implement all new operators.

2.5 SCSQ and SCSQL

SCSQ is a data stream management system built on top of Amos II with added facilities for querying of large volume, distributed streams through its high-level, declarative, continuous query language SCSQL [31, 28]. SCSQL is an extension of AmosQL but extended with parallel stream query facilities. The system is ported to many different platforms and currently runs in environments ranging from Windows to a massively parallel, 12000 node, IBM Bluegene cluster. Through SCSQL, high volume streams from distributed sources can be filtered, transformed and joined. Currently, SCSQ offers the highest published performance of the LR benchmark with an *L-rating* of 64, greatly outperforming all previously published results [28].

SCSQ was extended with our new *stream window* datatype and FCQL operators.

2.6 SQLFront

In [12], an SQL-to-AmosQL translator called SQLFront was developed to enable Amos II mediation through SQL. Large parts of the SQLFront parser can be reused in the construction of an automatic CQL-to-FCQL translator.

3 FCQL

In this section we will first give a high-level overview of the FCQL system and the available operator classes. Later, we will take a closer look and document each FCQL operator in detail.

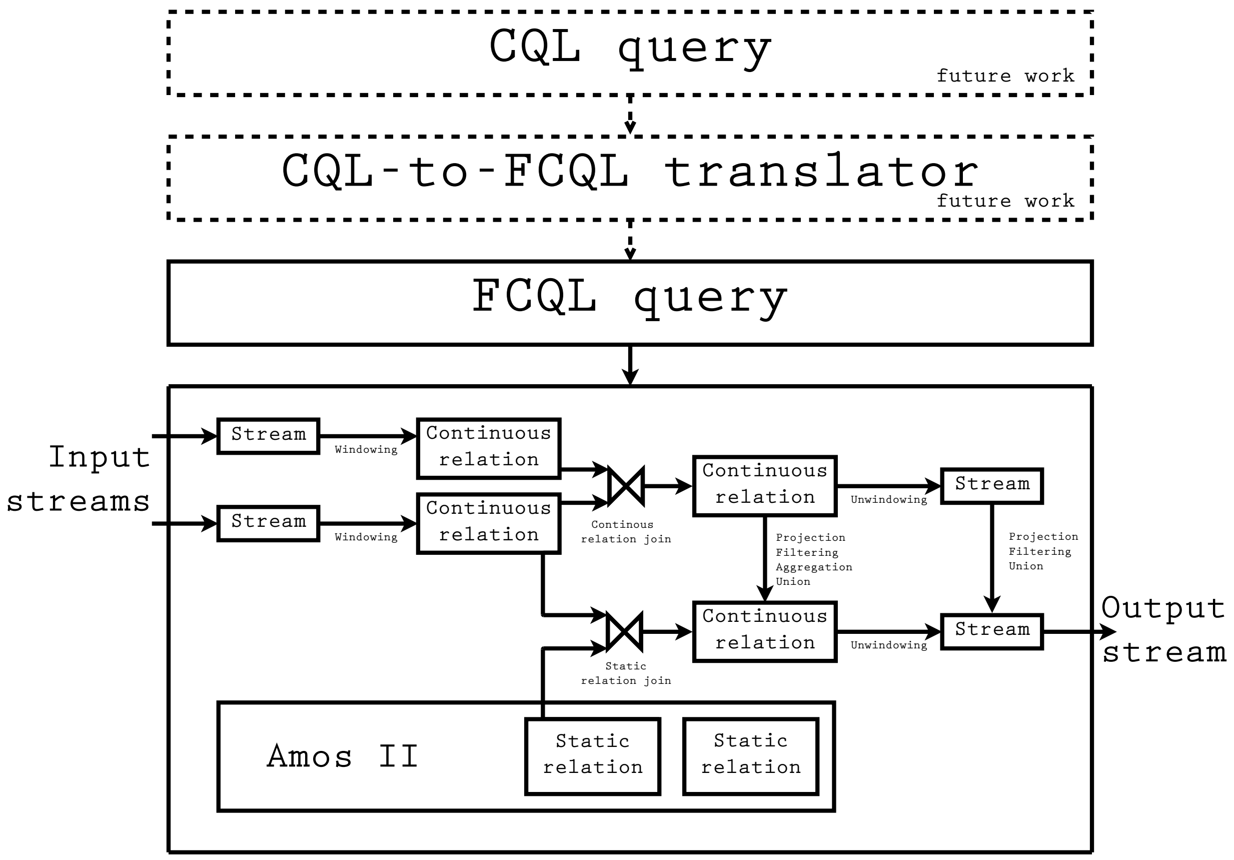

Figure 2: FCQL architecture.

Figure 2 shows the architecture of the FCQL system. FCQL extends AmosQL with continuous query primitives and has access to all facilities available in Amos II, including its main memory relational database. In the architecture overview, as part of Amos II, we can see *static relations* — those are regular relational tables represented as AmosQL functions.

Currently, continuous queries are specified as *FCQL queries*. However, it is planned future work to allow *CQL queries* and to translate those to FCQL queries using an automatic *CQL-to-FCQL translator*.

The FCQL system deals with three different types of data, *streams*, *continuous relations*, and *static relations*. Any number of streams can be used as input for the system. Through windowing operators streams are converted to continuous relations. Continuous relations can be manipulated using relational operators, such as projection, filtering, aggregation, union and joining. In the case of joining, continuous relations can either be joined among themselves, or a continuous relation can be joined with a static relation implemented as a function in the Amos II database. Continuous relations can be transformed back to streams using *unwindowing* operators. Some simple operations, such as projection and filtering, can be applied on streams without intermediate conversion to continuous relations.

3.2 Operator Classes

In FCQL, data operators are implemented as typed functions in the AmosQL query language. Below, the signature of each operator class is specified. Most operators are implemented as foreign functions. Furthermore, an extra AmosQL type *Window* has been introduced to represent stream windows.

The operators available in FCQL are the same as those available in the Stanford STREAM implementation of CQL, with one exception. We choose to implement a fourth class of operators which convert streams-to-streams. Our rationale is given in 3.2.

In FCQL, tuple streams are represented by the type *Stream of Vector*. The type *Vector* represents a single tuple. Continuously updated relations are represented by the type *Stream of Window*. The *Window* datatype is closely described in 3.3.4.

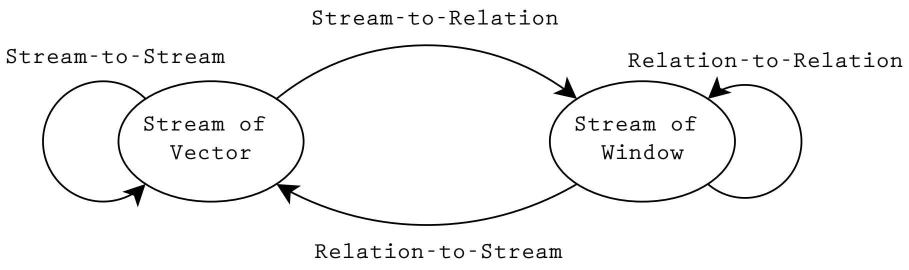

Figure 3: FCQL operator classes; stream-to-relation, relation-to-relation, relation-to-stream and stream-to-stream.

Stream-to-Relation Converts streams (*Stream of Vector*) to continuous relations (*Stream of Window*).

Relation-to-Relation Converts continuous relations (*Stream of Window*) to continuous relations (*Stream of Window*). This set of operators contains the operators usually found in SQL, such as projection, filtering, aggregation, joining, etc.

Relation-to-Stream Converts continuous relations (*Stream of Window*) to streams (*Stream of Vector*).

Stream-to-Stream These operators are not available in STREAM CQL. We choose to implement them as a convenience where it is unnecessary to convert a stream to a continuous relation for manipulation, and then immediately convert the relation back to a stream.

3.3 Operator Details

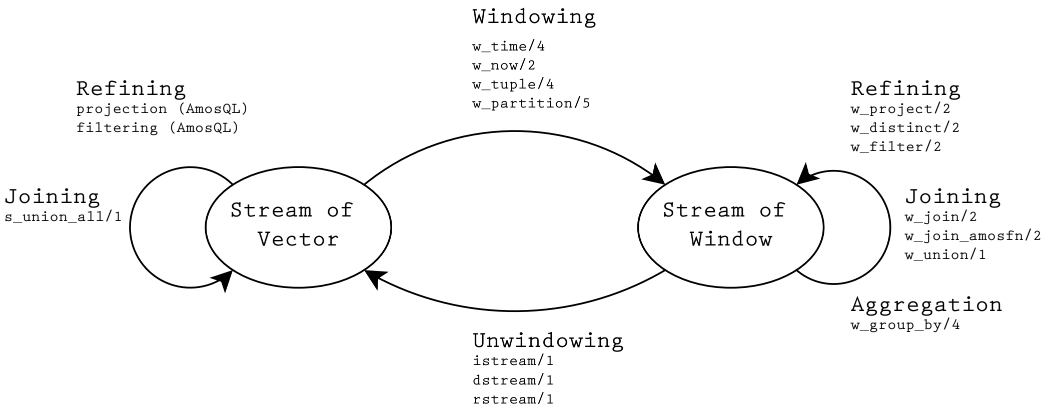

Figure 4: Detailed FCQL operator overview.

Figure 4 gives a detailed overview of the operators implemented as AmosQL functions available in FCQL. Stream-to-relation operators are labeled *windowing* and take streams (*Streams of Vectors*) as input and output continuous relations (*Streams of Windows*). They include *w_time/4* and *w_now/2* for time windows, *w_tuple/4* for counting windows, and *w_partition/5* for partitioned windows [17].

Relation-to-relation operators take continuous relations (*Streams of Windows*) as input and output continuous relations (*Streams of Windows*). They are divided into three classes of operators; the first class is labeled *Refining* and has operators for continuous relation projection and filtering. Those operators include *w_project/2*, *w_distinct/2* and *w_filter/2*. The second class of operators are *Joining*. They are used to combine continuous relations with each other or continuous relations with external data sources. Those operators include *w_join/2*, *w_join_amosfn/2* and *w_union/1*.

The last class of *Aggregation* operators includes a single operator, *w_group_by/4*, which is used to perform aggregation on continuous relations.

Relation-to-stream operators are labeled *unwindowing* and take continuous relations (*Streams of Windows*) as input and output streams (*Streams*

	CQL	FCQL
Windowing	Stream → Relation	Stream of Vector → Stream of Window
Relational	Relation → Relation	Stream of Window → Stream of Window
Unwindowing	Relation → Stream	Stream of Window → Stream of Vector
Streaming	— — — — — —	Stream of Vector → Stream of Vector

Table 1: How CQL operators relate to FCQL operators. As previously, the term *relation* refers to a continuous relation.

of Vectors). They include *istream/1*, *dstream/1* and *rstream/1* [17].

Stream-to-stream operators take streams (*Streams of Vectors*) for input and output streams (*Streams of Vectors*). They are divided into two classes; *Refining* and *Joining*. Streams are projected and filtered by existing AmosQL functionality. Our union all operator is defined by *s_union_all/1*.

Table 1 summarizes how all CQL operator classes relate to FCQL operators.

Starting with section 3.3.4, all FCQL operators will be documented together with examples showing how each operator corresponds to its STREAM CQL counterpart. However, to prepare for those examples, we will first document some common FCQL functionality; namely *FCQL schemas* and associative referencing of stream columns.

3.3.1 Schemas

In CQL, queries can be named and later used as sub-queries, corresponding to views in relational databases. Often this allows us to avoid duplicating code, and sometimes it even provides additional expressibility — for example where in-lined sub-queries are impossible.

In order to use a query as a sub-query it must have a schema. The schema will serve the same purpose as the schema of DBMS tables, defining the names and types of the query columns and naming views defined as queries.

In FCQL, a query is defined by an AmosQL *query function* which either produces a stream or a continuously updated relation. The query is named by the query functions' name, but the query function does not define a schema for its output. Instead, each query function is accompanied by a *query metadata function* which specifies the query schema by its signature. These metadata functions have the same name as the function they accompany, but are additionally prefixed by __meta__.

We will illustrate with an example. Given a *Customer* stream with the following schema:

```
Customer(id, name, age, ssn)
```

Listing 1: Stream of customers.

CQL Using the Stanford STREAM scripting language:

```
vquery: SELECT name, ssn FROM People WHERE age<18;
vtable: REGISTER STREAM Underage(name char(32), ssn integer);
```

Listing 2: Simple filtering on the *Customer* stream using the Stanford STREAM scripting language.

We can see that in the STREAM scripting language the query consists of two parts, the first, called *vquery* defines the actual query, while the second part, called *vtable*, defines its schema.

FCQL Using a FCQL metadata function:

```
    create function __meta__Underage() -> <Charstring name, Integer ssn>;
    create function Underage() -> Stream of Vector as select istream(...);
```

Listing 3: Simple filtering on the People stream using FCQL.

In FCQL the query is defined by the *Underage()* function. The signature of the *__meta__Underage()* metadata function defines the query schema. *__meta__Underage()* has no function body because its signature is sufficient to store the required metadata, i.e., the column names and types of the *Underage* stream. We also left out the function body of the *Underage()* function to simplify our example. Detailed query definitions are given in 3.3.4.

3.3.2 Column Referencing

Starting with section 3.3.4 we will see that FCQL operators reference elements of streams tuples by their positional index in a tuple. In other words, the operators do not expect the name of the column that should be manipulated as in CQL, but the index at which the column is found in the tuples of a stream or a continuously updated relation.

Since it is often desirable to reference columns associatively, i.e., by name, we offer three functions which translate a column name to a positional index; *res_idx/2*, *join_res_idx/2* and *join_res_idx/3*.

res_idx(Charstring fn, Charstring field) → Integer index implements the *'.'* operator in SQL/CQL to reference elements of tuples. The function takes a FCQL query name *fn* and the name *field* of one of the query's output fields. Returns the position among the query's output fields at which the field can be found in the query, the first field starts at index 0. If the field can't be found *nil* is returned.

Examples Using res_idx/2:

```
create function __meta__Underage() -> <Charstring name, Integer ssn>;

res_idx('Underage', 'name');
>> 0
res_idx('Underage', 'ssn');
>> 1
res_idx('Underage', 'age');
>> NIL
```

join_res_idx(Vector fnv, Charstring field) → <Integer fn_index, Integer col_index> is used to reference the fields of a joined continuous relation as created by *w_join/2* 3.3.5 or *w_join_amosfn/2* 3.3.5. It takes a vector of FCQL query names *fnv*, where each query is one of the joined continuous relations. The function determines the first query which contains the output field, searching *fnv* in order, and the index at which the output field *name* was found. Finally it returns a tuple where the first element *fn_index* is the index of the container query and the second element *col_index* is the index of the output field. If the field can't be found *nil* is returned.

Examples Using *join_res_idx/2*:

```
create function __meta__Parent() ->
  <Charstring name, Integer child_ssn, Integer ssn>;
create function __meta__Underage() ->
  <Charstring name, Integer ssn>;

join_res_idx({'Parent', 'Underage'}, 'name');
>> <0,0>
join_res_idx({'Parent', 'Underage'}, 'ssn');
>> <0,2>
join_res_idx({'Parent', 'Underage'}, 'child_ssn');
>> <0,1>
join_res_idx({'Parent', 'Underage'}, 'age');
>> NIL
```

join_res_idx(Vector fnv, Charstring fn_target, Charstring field) → <Integer function_index, Integer column_index> is just like *join_res_idx/2* but specifies with *fn_target* which of the queries in *fnv* to look in. If the field can't be found in the query *fn_target*, *nil* is returned.

Examples Using *join_res_idx/3*:

```
create function __meta__Parent() ->
  <Charstring name, Integer child_ssn, Integer ssn>;
create function __meta__Underage() -> <Charstring name, Integer ssn>;

join_res_idx({'Parent', 'Underage'}, 'Parent', 'name');
>> <0,0>
join_res_idx({'Parent', 'Underage'}, 'Parent', 'ssn');
>> <0,2>
join_res_idx({'Parent', 'Underage'}, 'Parent', 'child_ssn');
>> <0,1>
join_res_idx({'Parent', 'Underage'}, 'Parent', 'age');
>> NIL

join_res_idx({'Parent', 'Underage'}, 'Underage', 'name');
>> <1,0>
join_res_idx({'Parent', 'Underage'}, 'Underage', 'ssn');
>> <1,1>
join_res_idx({'Parent', 'Underage'}, 'Underage', 'child_ssn');
>> NIL
join_res_idx({'Parent', 'Underage'}, 'Underage,␣'age');
>> NIL
```

3.3.3 Input Streams

In the following sections all FCQL operators will be documented, and usage examples given that show how FCQL operators relate to CQL operators. Throughout those examples the following relations and input streams will be used:

```
Transaction(time, transaction_id, sender, receiver, amount);
```

Listing 4: A stream of bank transactions.

In FCQL, the Transaction stream would be defined in the following way:

```
create function __meta__Transaction() ->
  <Integer name,
   Integer transaction_id,
   Integer sender,
   Integer receiver,
   Integer amount>;
create function Transaction() -> Stream of Vector as select streamof(...);
```

Listing 5: Transaction stream in FCQL.

The remaining streams are defined analogously.

```
TransactionVISA(time, transaction_id, sender, receiver, amount);
```

Listing 6: A stream of VISA transactions.

```
BalanceQuery(time, query_id, user_id);
```

Listing 7: A stream of account balance queries.

```
User(user_id, name);
```

Listing 8: A static relation of bank account owners.

3.3.4 Stream-to-Relation Operators

As in STREAM CQL, the stream-to-relation operators in FCQL are based on a *sliding window* and produce continuously updated relations. Unlike the STREAM implementation, where all windows have their slide set to 1 [17], FCQL allows the slide to be defined by the user for all windowing operators. Furthermore, in STREAM CQL, it is assumed that the first column of each tuple is its timestamp. In FCQL, the timestamp column is user defined for all window operators.

Continuous relations in FCQL are represented by a stream of windows where each subsequent window holds a new state of the continous relation. The window datatype is made up from a linked list of tuples and some metadata as seen in figure 5.

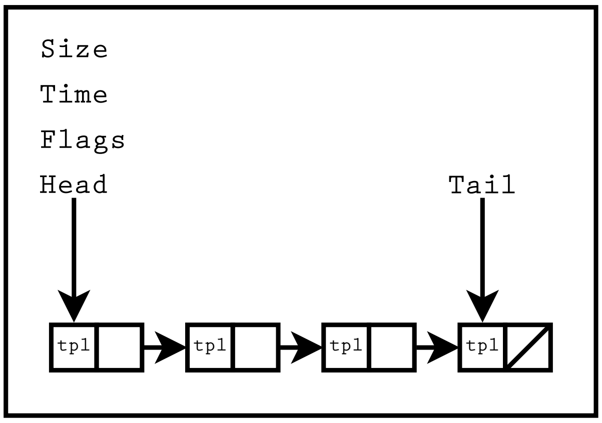

Figure 5: The *window* datatype contains a linked list of tuples and keeps track of the list's *head* for removal of tuples, the list's *tail* for addition of tuples, a window *size*, a window *time* field used by the time window operator, and a *flags* field.

w_time(Stream of Vector s, Integer size, Integer slide, Integer ts_idx) → Stream of Window wstream is our time window construction operator. It takes a stream of vectors (tuples) s, a window size *size*, a window slide *slide*, and a index *ts_idx* that specifies the column at which a timestamp can be found in the vectors in s. The function returns a stream of windows where the timestamp distance from the first to last tuple in each window is at most *size* time units large, and where each consequent window has been advanced *slide* time units (dropping the oldest tuples from the previous window).

Examples The time window in CQL and FCQL:

```
SELECT * FROM Transaction [RANGE 10 SECONDS] ;
```

Listing 9: CQL time window highlighted in green.

```
w_time(Transaction(), 10, 1, res_idx('Transaction', 'time'));
```

Listing 10: FCQL time window.

w_now(Stream of Vector s, Integer ts_idx) → Stream of Window wstream is a convenience operator that behaves like *w_time* but has its *size* and *slide* set to 1.

Examples The time window in CQL and FCQL:

```
SELECT * FROM Transaction [NOW];
```

Listing 11: CQL now window highlighted in green.

```
w_now(Transaction(), res_idx('Transaction', 'time'));
```

Listing 12: FCQL now window.

w_tuple(Stream of Vector s, Integer size, Integer slide, Integer ts_idx) → Stream of Window wstream is our counting window construction operator. It takes a stream of vectors (tuples) *s*, a window size *size*, a window slide *slide* and a index *ts_idx* which specifies the column at which a timestamp can be found in the vectors in *s*. Unlike *w_time*, where the *size* and *slide* are measured in time units, *w_tuple* treats them as an exact number of tuples. The operator returns a stream to windows where each window is at most *size* tuples large and each consequent window has been slided forward by *slide* tuples (dropping the *slide* oldest tuples from the previous window).

Note that we require a timestamp index *ts_idx* in our counting window operator. CQL uses a time-driven model while processing tuples [17]. For windowing operators, including the counting window, this implies that the output is produced at the end of each timestamp, i.e., when all tuples for any given timestamp have been seen. When a timestamp is such that there are more input tuples than can be placed inside the window, without violating its maximum window size, the window will only keep the most recent input tuples. Hence, in order for the window operator to determine when a timestamp ends, it must be aware of the tuple timestamp index.

Examples The counting window in CQL and FCQL:

```
SELECT * FROM Transaction [ROWS 10] ;
```

Listing 13: CQL counting window highlighted in green.

```
w_tuple(Transaction(), 10, 1, res_idx('Transaction', 'time'));
```

Listing 14: FCQL counting window.

w_partition(Stream of Vector s, Vector idx, Integer size, Integer slide, Integer ts_idx) → Stream of Window wstream is our partitioned window construction operator. It takes a stream of vectors (tuples) *s*, a vector of indexes *idx* that specify the columns in *s* on which to perform partitioning, a window size *size*, a window slide *slide*, and an index *ts_idx* that specifies the column at which a timestamp can be found in the vectors in *s*. The timestamp index is required for the same reason as in the counting window 3.3.4. Just as in *w_tuple*, *size* and *slide* are treated as an exact number of tuples. The operator returns a stream of windows where each window is constructed by partitioning the stream *s* on the columns in *idx*, forming one counting window for each partition. When one of the partitions is filled, the tuples of all partitions are unified into one window and it is emitted. The next window will have all its partitions slided forward by *slide*.

Examples The partition window in CQL and FCQL:

```
SELECT * FROM Transaction [PARTITION BY seller ROWS 10] ;
```

Listing 15: CQL partition window highlighted in green.

```
w_partition(Transaction(), {res_idx('Transaction', 'seller')},
            10, 1, res_idx('Transaction', 'time'));
```

Listing 16: FCQL partition window.

3.3.5 Relation-to-Relation Operators

The largest part of the data manipulation of a CQL query is performed by relation-to-relation operators [2, 1, 30, 17]. The following relation-to-relation operators are available in FCQL:

w_project(Vector idx, Stream of Window wstream) → Stream of Window wstream takes a vector of indexes *idx* that specify which columns in the window stream *wstream* to project. It returns a new stream of windows where only the projected columns remain in the tuples of the windows in the window stream *wstream*.

Examples Continuous relation projection in CQL and FCQL:

```
SELECT time, transaction_id, amount
FROM Transaction [ROWS 10];
```

Listing 17: CQL projection highlighted in green.

```
w_project(
  {res_idx('Transaction', 'time'),
   res_idx('Transaction', 'transaction_id'),
   res_idx('Transaction', 'amount')} ,
  w_tuple(Transaction(), 10, 1, res_idx('Transaction', 'time')));
```

Listing 18: FCQL projection highlighted in green.

w_distinct(Vector idx, Stream of Window wstream) → Stream of Window wstream takes a vector of indexes *idx* that specify which columns to use when forming a uniqueness key for the tuples in the window stream *wstream*. The function returns a new window stream where each window is internally unique. When duplicates are found, the first tuple encountered will remain and all subsequent tuples discarded.

Examples Distinct operator on continuous relations in CQL and FCQL:

```
SELECT DISTINCT sender , receiver , amount
FROM Transaction [ROWS 10];
```

Listing 19: CQL distinct operator highlighted in green.

```
w_distinct(
  {res_idx('Transaction', 'sender')},
  w_project(
    {res_idx('Transaction', 'receiver'),
     res_idx('Transaction', 'amount')},
    w_tuple(Transaction(), 10, 1, res_idx('Transaction', 'time')));
```

Listing 20: FCQL distinct operator highlighted in green.

w_filter(Stream of Window wstream, Function accept_fn) → Stream of Window wstream takes a stream *wstream* of windows and an accept function *accept_fn* with the signature *accept_fn(Vector tuple) → Boolean. accept_fn* is given tuples from *wstream* and should return *true* for each tuple that should not be discarded.

The function returns a new window stream where only accepted tuples remain.

Examples Filtering on continuous relations in CQL and FCQL:

```
SELECT time, receiver, amount
FROM Transaction [ROWS 10]
WHERE amount > 50;
```

Listing 21: CQL filtering highlighted in green.

```
create function __filter__Transaction(Vector tuple) -> Boolean as
  tuple[res_idx('Transaction', 'amount')] > 50;

w_project(
  {res_idx('Transaction', 'time'),
   res_idx('Transaction', 'receiver'),
   res_idx('Transaction', 'amount')},
  w_filter(
    w_tuple(Transaction(), 10, 1, res_idx('Transaction', 'time')),
    #'__filter__Transaction'));
```

Listing 22: FCQL filtering with the filtering function *__filter__Transaction/1* highlighted in green. The filtering function is given each tuple of the continuous relation *Transaction()* and will return *true* for all tuples where *amount* is larger than 50.

w_join(Vector wstream, Function join_fn) → Stream of Window wstream takes a vector of window streams *wstream* that should be n-theta joined based on an variadic joining function *join_fn* with the signature *join_fn(Vector 1, Vector 2, ..., Vector n) → Boolean.* The joining function is given *n* arguments, where each argument *i* is a tuple taken from window stream *i* in *wstream*, and should return *true* if the tuples are joinable.

The function returns a new window stream where the windows of *wstream* have been joined.

Examples Joining of continuous relations in CQL and FCQL:

```
SELECT B.time, T.sender, T.receiver, T.amount
FROM BalanceQuery [NOW] AS B, Transaction [RANGE 30 DAYS] AS T
WHERE B.user_id = T.receiver OR
      B.user_id = T.sender;
```

Listing 23: CQL joining highlighted in green.

```
create function __join__TransactionBalance(Vector B, Vector T) -> Boolean as
  B[res_idx('BalanceQuery', 'user_id')] =
  T[res_idx('Transaction', 'receiver')] OR
  B[res_idx('BalanceQuery', 'user_id')] =
  T[res_idx('Transaction', 'sender')];

w_project(
  {join_res_idx({'BalanceQuery', 'Transaction'}, 'time'),
   join_res_idx({'BalanceQuery', 'Transaction'}, 'sender'),
   join_res_idx({'BalanceQuery', 'Transaction'}, 'receiver'),
   join_res_idx({'BalanceQuery', 'Transaction'}, 'amount')},
  w_join(
    {w_now(BalanceQuery(), res_idx('BalanceQuery', 'time')),
     w_time(Transaction(), 30*24*60*60, 1, res_idx('Transaction', 'time'))},
    #'__join__TransactionBalance'));
```

Listing 24: FCQL joining with the joining function *__join__TransactionBalance/2* highlighted in green.

w_join_amosfn(Stream of Window wstream, Function amosfn) → Stream of Window wstream is used to join continuous relations with static relations. The function takes a window stream *wstream* that should be joined with the AmosQL function *amosfn*, which has the signature *amosfn(Vector* *tuple)* → *Bag of Vector*. The AmosQL function is given each tuple *t* from *wstream*, and should for each *t* return a bag of tuples which are joinable with *t*.

w_join_amosfn_2 returns a stream of windows where each window is created by joining each window of *wstream* with *amosfn*.

Examples Joining a continuous relation with a static relation in CQL and FCQL:

```
SELECT B.query_id, B.user_id, U.name
FROM BalanceQuery [NOW] AS B, User AS U
WHERE B.user_id = U.user_id;
```

Listing 25: CQL joining with a static relation highlighted in green.

```
create function __join_amosfn__BalanceUser(Vector tuple) -> Bag of Vectoras
  select other_tuple
  from Vector other_tuple
  where other_tuple in streamof(User()) and
  tuple[res_idx('BalanceQuery', 'user_id')] =
  other_tuple[res_idx('User', 'user_id')];

w_project(
  {join_res_idx({'BalanceQuery', 'User'}, 'query\_id'),
   join_res_idx({'BalanceQuery', 'User'}, 'user\_id'),
   join_res_idx({'BalanceQuery', 'User'}, 'name')},
  w_join_amosfn(
    w_now(BalanceQuery(), res_idx('BalanceQuery', 'time')),
    #'__join_amosfn__BalanceUser'));
```

Listing 26: FCQL joining with the static relation *User()* highlighted in green. The joining function *__join_amosfn__BalanceUser/1* is given each tuple of the continuous relation *BalanceQuery()* as *tuple* and will return all joinable tuples from the relation *User()*.

w_union(Vector wstream) → Stream of Window wstream takes a vector of window streams *wstream* and forms a union of the windows where one window each is taken from each stream. The elements of each union formed this way is emitted as a window in a new window stream. Unlike *w_join/2*, which stops if one window stream runs out of windows, *w_union/1* continues until all window streams are exhausted.

Examples Union of continuous relations in CQL and FCQL:

```
SELECT * FROM Transaction [ROWS 10]
UNION
SELECT * FROM TransactionVISA [ROWS 10];
```

Listing 27: CQL union highlighted in green.

```
w_union({
  w_tuple(Transaction(), 10, 1, res_idx('Transaction', 'time')),
  w_tuple(TransactionVISA(), 10, 1, res_idx('TransactionVISA', 'time'))});
```

Listing 28: FCQL union highlighted in green.

w_group_by(Vector key_idx, Vector val_idx, Vector agg_fn, Stream of Window wstream) → Stream of Window wstream groups each window in the stream *wstream* based on keys in vector *key_idx* and applies the aggregate functions in *agg_fn* on the corresponding columns of *val_idx*. The result of the function is a stream of windows where each window is made up of the calculated aggregate values.

Examples Continuous relation aggregation in CQL and FCQL:

```
SELECT user_id, SUM(amount)
FROM Transaction [RANGE 30 DAYS]
GROUP BY user_id;
```

Listing 29: CQL aggregation highlighted in green.

```
create function __meta__GroupbyTransaction() ->
  <Integer user_id, Integer amount>;

w_project(
  {res_idx('GroupbyTransaction', 'user_id'),
   res_idx('GroupbyTransaction', 'amount')},
  w_group_by(
    {res_idx('Transaction', 'user_id')},
    {res_idx('Transaction', 'amount')},
    {#'sum'},
    w_time(Transaction(), 30*24*60*60, 1, res_idx('Transaction', 'time')));
```

Listing 30: FCQL aggregation highlighted in green. The metadata function *__meta__GroupbyTransaction/0* defines the schema of the group by operators' output.

3.3.6 Relation-to-Stream Operators

When continuous relations (*Streams of Window*) must be converted back to streams, the operators *istream/1*, *dstream/1*, *rstream/1* can be used. First a formal definition is given [2]:

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

The operators are documented bellow:

istream(Stream of Window wstream) → Stream of Vector s takes a stream of windows and produces a stream of those tuples added to the window stream. A tuple is added when it exists in one window but not in the previous.

Examples *Istream* operator in CQL and FCQL:

```
ISTREAM( SELECT *
         FROM Transaction [ROWS 10]);
```

Listing 31: CQL istream operator highlighted in green.

```
istream(
  w_tuple(Transaction(), 10, 1, res_idx('Transaction', 'time')));
```

Listing 32: FCQL istream operator highlighted in green.

dstream(Stream of Window wstream) → Stream of Vector s takes a stream of windows and produces a stream of those tuples removed to the window stream. A tuple is removed when it exists in one window but not in the next.

Examples *Dstream* operator in CQL and FCQL:

```
DSTREAM( SELECT *
         FROM Transaction [ROWS 10]);
```

Listing 33: CQL dstream operator highlighted in green.

```
dstream(
  w_tuple(Transaction(), 10, 1, res_idx('Transaction', 'time')));
```

Listing 34: FCQL dstream operator highlighted in green.

rstream(Stream of Window wstream) → Stream of Vector s takes a stream of windows and produces a stream of all tuples in each window.

Examples *Rstream* operator in CQL and FCQL:

```
RSTREAM( SELECT *
         FROM Transaction [ROWS 10]);
```

Listing 35: CQL rstream operator highlighted in green.

```
rstream(
  w_tuple(Transaction(), 10, 1, res_idx('Transaction', 'time')));
```

Listing 36: FCQL rstream operator highlighted in green.

3.3.7 Stream-to-Stream Operators

As previously mentioned, the STREAM CQL implementation has no stream-to-stream operators and all data manipulation is performed by relation-to-relation operators; in FCQL, we can manipulate streams directly.

Stream projection and filtering is handled by existing AmosQL functionality.

union_all(Vector s) → Stream of Vector s takes a vector of tuple streams *s* and outputs a new tuple stream such that tuples are taken from *s* in order and as they become available. As *w_union/1*, *union_all* terminates only when all input streams are exhausted.

Examples Union all on streams in CQL and FCQL:

```
SELECT * FROM Transaction
UNION ALL
SELECT * FROM TransactionVISA;
```

Listing 37: CQL union all highlighted in green. Note that this query implicitly converts the *Transaction* and *TransactionVISA* streams to continuous relations and then back to streams.

```
s_union_all({
  Transaction(),
  TransactionVISA()});
```

Listing 38: FCQL union all highlighted in green.

4 Conclusions and Future Work

We extended SCSQ with the functional continuous query language FCQL and implemented all operators available in STREAM CQL. Furthermore, in appendix A we show that FCQL is robust enough to express all CQL queries in the STREAM implementation of the LR benchmark. We have also shown how CQL operators and queries can be translated to FCQL and that this process is systematic and mechanic enough to be well suited for an automatic CQL-to-FCQL translator.

Although FCQL has the same expressive power as CQL, CQL queries are easier to write, read and understand. Therefore, future work involves a parser that translates CQL queries to FCQL. Syntactically, CQL is very similar to SQL and large parts of the SQLFront parser [12] can be reused by a CQL parser.

Since CQs run indefinitely — or until they are terminated — and queries are often used as sub-queries, there is often more than one query using any given query as a sub-query. In such a scenario the sub-query should be able to feed its output to all its consumer queries. In FCQL, a query may only have a single consumer. This problem can be solved by using SCSQ *stream processes* [31] and *splitstream functions* [32]; queries that have more than one consumer should execute in separate SCSQ stream processes and disseminate their output to all consumers using splitstream functions.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. *PVLDB*, 3(1), 2010.

[6] Brandeis University. Linear Road Benchmark Homepage. `http://pages.cs.brandeis.edu/~linearroad/index.html`.

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

[9] Guy L. Steele. Common Lisp the Language, 2nd edition. `http://www.ida.liu.se/imported/cltl/cltl2.html`.

[10] Timour Katchaounov, Tore Risch, and Simon Zürcher. Object-oriented mediator queries to internet search engines. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information*

Systems, volume 2426 of *Lecture Notes in Computer Science*, pages 241–246. Springer Berlin / Heidelberg, 2002.

[11] Peter Lyngbæk. OSQL: A language for object databases. Technical report, Palo Alto, California, HPL-DTD-91-4, 1991.

[12] Markus Jägerskogh. Translating SQL expressions to Functional Queries in a Mediator Database System. `http://user.it.uu.se/~udbl/Theses/MarkusJagerskoghMSc.pdf`.

[13] Mattias Nyström and Tore Risch. Optimising mediator queries to distributed engineering systems. In YoonJoon Lee, Jianzhong Li, Kyu-Young Whang, and Doheon Lee, editors, *Database Systems for Advanced Applications*, volume 2973 of *Lecture Notes in Computer Science*, pages 193–209. Springer Berlin / Heidelberg, 2004.

[14] Tore Risch. Functional queries to wrapped educational semantic web meta-data.

[15] Tore Risch and Vanja Josifovski. Distributed Data Integration By Object-Oriented Mediator Servers, 2001.

[16] Tore Risch, Vanja Josifovski, and Timour Katchaounov. Functional Data Integration in a Distributed Mediator System, 2003.

[17] Robert Kajic. Evaluation of the Stream Query Language CQL. `http://user.it.uu.se/~udbl/Theses/RobertKajicBSc.pdf`.

[18] David W. Shipman. The functional data model and the data language daplex. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 59–59, New York, NY, USA, 1979. ACM.

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

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

[21] StreamBase. StreamBase Homepage. `http://www.streambase.com/`.

[22] Tore Risch. ALisp v2 User's Guide.

[23] Tore Risch. Amos II External Interfaces.

[24] Uppsala DataBase Laboratory. Amos II Concepts. http://user.it.uu.se/~udbl/amos/doc/amos_concepts.html.

[25] Uppsala DataBase Laboratory. Amos II Homepage. http://user.it.uu.se/~udbl/amos/.

[26] Uppsala DataBase Laboratory. Amos II Wrappers. http://user.it.uu.se/~udbl/amos/wrappers.html.

[27] Uppsala DataBase Laboratory. AmosQL User's Manual. http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html.

[28] Uppsala DataBase Laboratory. SCSQ Homepage. http://user.it.uu.se/~udbl/SCSQ.html.

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

[30] 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.

[31] Erik Zeitler and Tore Risch. Using stream queries to measure communication performance of a parallel computing environment. *Distributed Computing Systems Workshops, International Conference on*, 0:65, 2007.

[32] Erik Zeitler and Tore Risch. Scalable splitting of massive data streams. In Hiroyuki Kitagawa, Yoshiharu Ishikawa, Qing Li, and Chiemi Watanabe, editors, *Database Systems for Advanced Applications*, volume 5982 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin / Heidelberg, 2010.

A Evaluation

To evaluate the completeness of FCQL we choose to translate the complete LR benchmark specification as it is defined by the Stanford STREAM project in [19]. The benchmark queries are translated in the same order as they are defined in [19] and each translation is preceded by the original CQL query together with explanatory comments. With the help of these translations we will show that the translation process is systematic and mechanic. Furthermore, the LR benchmark is complicated enough to make use of all FCQL operators and makes these translations well suited to be used as a template for an automatic CQL-to-FCQL translator. Finally, they serve as a compliment to the FCQL documentation in 3.3.

```
/***********************************************************
 *
 *  AMOS2
 *
 *  Author: (c) 2010 Robert Kajic, UDBL
 *  $RCSfile: lr.osql,v $
 *  $Revision: 1.14 $ $Date: 2010/10/18 14:52:42 $
 *  $State: Exp $ $Locker:  $
 *
 *  Description: Translations of the linear road benchmark
 *  specification as specified in CQL by the Stanford STREAM
 *  project. The CQL specification can be found at:
 *  http://infolab.stanford.edu/stream/cql-benchmark.html
 *  ==========================================================
 *  $Log: lr.osql,v $
 *  Revision 1.14  2010/10/18 14:52:42  roka4241
 *  *** empty log message ***
 *
 *  Revision 1.13  2010/10/13 02:25:15  roka4241
 *  Changed the n-theta join to call its joining funtion with n arguments instead of a vector wit
 *
 *  Revision 1.12  2010/10/12 13:08:20  roka4241
 *  Removed s_project and s_filter, using AmosQL functions instead.
 *
 *  Revision 1.11  2010/10/08 02:33:27  roka4241
 *  res_idx and join_res_idx now take the name of the query function for which to resolve an outp
 *
 *  Revision 1.10  2010/09/07 23:31:42  roka4241
 *  Removed old regression tests. Fixed conflicts in lr.osql.
 *
 *  Revision 1.9  2010/09/07 23:25:34  roka4241
 *  Started using s_project.
 *
 *  Revision 1.8  2010/08/22 03:47:24  roka4241
 *  Started using amos_join_fn.
 *
 *  Revision 1.7  2010/08/10 02:57:11  roka4241
 *  Fixed all syntax errors. Added some forgotten functions. Fixed some bugs, mostly typo related
 *
 *
 *  Revision 1.5  2010/08/07 15:25:36  roka4241
 *  Translated the remaining lr queries from the cql specification. Fixed some bugs.
 *
 *  Revision 1.4  2010/08/07 04:10:45  roka4241
 *  Added this CVS header.
 *
 *
 **************************************************************/

/*
parteval("res_idx");
parteval("join_res_idx");
*/

/* Static amos function that returns all segments. Currently, the function
returns a static bag of segments but it should eventually compute the bag
of segments based on the size of the LR benchmark being run. */
create function __meta__AllSeg() ->
  <Integer exp_way, Integer dir, Integer seg>;
create function AllSeg() -> Bag of Vector as select
  bag(
    {0, 0, 0},
    {0, 0, 1},
    {0, 1, 0},
    {0, 1, 1},
    {1, 0, 0},
    {1, 0, 1},
    {1, 1, 0},
    {1, 1, 1});

/* The SegToll and AccAffectedSeg streams are used before they are
defined and we must define stub functions to satisfy the compiler. */

create function __meta__SegToll() ->
  <Integer time, Integer exp_way, Integer dir, Integer seg, Integer toll>;
create function SegToll() -> Stream of Window;

create function __meta__AccAffectedSeg() ->
  <Integer time, Integer exp_way, Integer dir, Integer seg>;
create function AccAffectedSeg() -> Stream of Window;

/*********
 *********
 ********* INPUT
 *********
 *********/

/* CarLocStr: Stream of car location reports. This forms primary input to the system.

CarLocStr(car_id,          unique car identifier
          speed,           speed of the car
          exp_way,         expressway: 0..10
          lane,            lane: 0,1,2,3
          dir,             direction: 0(east), 1(west)
          x-pos);          coordinate in express way */

create function __meta__CarLocStr() ->
  <Integer time, Integer car_id, Integer speed,
   Integer exp_way, Integer lane, Integer dir, Integer x_pos>;

create function CarLocStr() -> Stream of Vector as select streamof(
  select vector(v[1], v[2], v[3], v[4], v[5], v[6], v[8])
  from Vector v
  where v in streamfile("data/lr32")
  and v[0] = 0);
```

```
/* AccBalQueryStr: Stream of account-balance adhoc queries. Each query
requests the current account balance of a car.

AccBalQueryStr(car_id,
               query_id); id used to associate responses with queries */

create function __meta__AccBalQueryStr() ->
  <Integer time, Integer car_id, Integer query_id>;

create function AccBalQueryStr() -> Stream of Vector as select streamof(
  select vector(v[1], v[9], v[10])
  from Vector v
  where v in(streamfile("data/lr32"))
  and v[0] = 1);

/* ExpQueryStr: Stream of adhoc queries requesting the expenditure of
a car for the current day.

ExpQueryStr(car_id,
            query_id); */
create function __meta__ExpQueryStr() ->
       <Integer time, Integer car_id, Integer query_id>;

create function ExpQueryStr() -> Stream of Vector as select streamof(
  select vector(v[1], v[11], v[12])
  from Vector v
  where v in(streamfile("data/lr32"))
  and v[0] = 2);

/* TravelTimeQueryStr: Stream of expected-travel-time adhoc queries.

TravelTimeQueryStr(query_id,
                   exp_way,
                   init_seg,        initial segment
                   fin_seg,         final segment
                   time_of_day,
                   day_of_week); */
create function __meta__CreditStr() ->
  <Integer time, Integer car_id, Integer query_id>;

create function CreditStr() -> Stream of Vector as select streamof(
  select vector(v[1], v[13], v[14])
  from Vector v
  where v in(streamfile("data/lr32"))
  and v[0] = 3);


/*********
 *********
 ********* TOLL NOTIFICATION
 *********
 *********/

/* CarSegStr (stream): This is just the input CarLocStr stream, but
with the location of the car replaced by the segment corresponding to
the location. For simplicity, we assume that each segment is
equi-length, and therefore the segment number for a location is just
the location (integer-)divided by the length of each segment.

SELECT car_id, speed, exp_way, lane, dir, (x-pos/52800) as seg
```

```
FROM CarLocStr; */

create function __meta__CarSegStr() ->
  <Integer time, Integer car_id, Integer speed,
   Integer exp_way, Integer lane, Integer dir,
   Integer seg>;

create function CarSegStr() -> Stream of Vector as select streamof(
   select
     {v[res_idx("CarLocStr", "time")],
      v[res_idx("CarLocStr", "car_id")],
      v[res_idx("CarLocStr", "speed")],
      v[res_idx("CarLocStr", "exp_way")],
      v[res_idx("CarLocStr", "lane")],
      v[res_idx("CarLocStr", "dir")],
      floor(cast(v[res_idx("CarLocStr", "x_pos")] as Number) / 52800)}
   from Vector v
   where v in CarLocStr());

/* CurActiveCars (relation): The relation containing the set of cars
"currently" on the freeway: cars that have reported their location in
the last 30 seconds.

SELECT DISTINCT car_id
FROM CarSegStr [RANGE 30 SECONDS]; */

create function __meta__CurActiveCars() ->
  <Integer time, Integer car_id>;

create function CurActiveCars() -> Stream of Window as select
  w_distinct(
    {res_idx("CurActiveCars", "car_id")},
    w_project(
      {res_idx("CarSegStr", "time"),
       res_idx("CarSegStr", "car_id")},
      w_time(CarSegStr(), 30, 1, res_idx("CarSegStr", "time"))));

/* CurCarSeg (relation): The relation containing the current segment
for each car on the freeway. This relation is obtained by computing
the latest segment for all the cars (the partition window below), and
picking only those cars that are currently active (by joining with
CurActiveCars relation above). Note that a segment is identified by
the expressway, the direction of travel, and the segment number within
the expressway.

SELECT car_id, exp_way, dir, seg
FROM CarSegStr [PARTITION BY car_id ROWS 1], CurActiveCars
WHERE CarSegStr.car_id = CurActiveCars.car_id; */

create function __meta__CurCarSeg() ->
  <Integer time, Integer car_id, Integer exp_way, Integer seg>;

create function __join__CurCarSeg(Vector seg, Vector car) -> Boolean as
  seg[res_idx("CarSegStr", "car_id")] = car[res_idx("CurActiveCars", "car_id")];

create function CurCarSeg() -> Stream of Window as select
  w_project(
    {join_res_idx({"CarSegStr", "CurActiveCars"}, "time"),
     join_res_idx({"CarSegStr", "CurActiveCars"}, "car_id"),
     join_res_idx({"CarSegStr", "CurActiveCars"}, "exp_way"),
     join_res_idx({"CarSegStr", "CurActiveCars"}, "dir"),
     join_res_idx({"CarSegStr", "CurActiveCars"}, "seg")},
    w_join(
      {w_partition(CarSegStr(), {res_idx("CarSegStr", "car_id")}, 1, 1,
                   res_idx("CarSegStr", "time")),
       CurActiveCars()},
      #'__join__CurCarSeg'));

/* CarSegEntryStr (stream): Stream of tuples each corresponding to
entry of a car into a new segment. A car enters a new segment when a
new entry appears in the CurCarSeg relation.

ISTREAM (CurCarSeg);

Some subtle issues arise when a car exits a segment, and re-enters the
same segment; unless we make some additional assumptions not specified
in the original benchmark specification, we cannot distinguish between
the case of a car continuing in a segment, and that of the car exiting
and subsequently re-entering the same segment. Presently, we assume
that a car that has exited remains outside the freeway network for
atleast 30 seconds, which enables the system to distinguish between
the two cases above. */

create function __meta__CarSegEntryStr() ->
  <Integer time, Integer car_id, Integer exp_way, Integer seg>;

create function CarSegEntryStr() -> Stream of Vector as select
  istream(CurCarSeg());

/* TollStr (stream): Stream of tolls. This is one of the output
streams of the benchmark. Each car, on entering a segment, pays a toll
determined by the current state of traffic in the segment. The
formulation of TollStr below uses a relation, SegToll, that contains
the current toll for each segment. The SegToll relation is specified
later.

SELECT RSTREAM(E.car_id, T.toll)
FROM CarSegEntryStr [NOW] AS E, SegToll as T
WHERE E.exp_way = T.exp_way AND E.dir = T.dir AND E.seg = T.seg;

The toll notification stream does not incorporate the
frequent-traveler discount as required by the original benchmark
specification. Doing so involves "recursion" (since the tolls depend
on the previous tolls), and CQL currently does not handle
recursion. */

create function __meta__TollStr() ->
  <Integer time, Integer car_id, Integer toll>;

create function __join__TollStr(Vector entry, Vector toll) -> Boolean as
  entry[res_idx("CarSegEntryStr", "exp_way")] = toll[res_idx("SegToll", "exp_way")] and
  entry[res_idx("CarSegEntryStr", "dir")] = toll[res_idx("SegToll", "dir")] and
  entry[res_idx("CarSegEntryStr", "seg")] = toll[res_idx("SegToll", "seg")];

create function TollStr() -> Stream of Vector as select
  rstream(w_project(
    {join_res_idx({"CarSegEntryStr", "SegToll"}, "CarSegEntryStr", "time"),
     join_res_idx({"CarSegEntryStr", "SegToll"}, "CarSegEntryStr", "car_id"),
     join_res_idx({"CarSegEntryStr", "SegToll"}, "SegToll", "toll")},
    w_join(
      {w_now(CarSegEntryStr(), res_idx("CarSegEntryStr", "time")),
       SegToll()},
      #'__join__TollStr')));
```

```
/*********
 *********
 ********* TOLL COMPUTATION FOR SEGMENTS
 *********
 *********/

/* SegAvgSpeed (relation) : The average speed of the cars in a segment
over the last 5 minutes.

SELECT exp_way, dir, seg, AVG(speed) as speed,
FROM CarSegStr [RANGE 5 MINUTES]
GROUP BY exp_way, dir, seg; */

create function __meta__SegAvgSpeed() ->
  <Integer exp_way, Integer dir, Integer seg, Integer speed>;

create function __meta__GroupbySegAvgSpeed() ->
  <Integer exp_way, Integer dir, Integer seg, Integer speed>;

create function SegAvgSpeed() -> Stream of Window as select
  w_project(
    {res_idx("GroupbySegAvgSpeed", "exp_way"),
     res_idx("GroupbySegAvgSpeed", "dir"),
     res_idx("GroupbySegAvgSpeed", "seg"),
     res_idx("GroupbySegAvgSpeed", "speed")},
    w_group_by(
      {res_idx("CarSegStr", "exp_way"),
       res_idx("CarSegStr", "dir"),
       res_idx("CarSegStr", "seg")},
      {res_idx("CarSegStr", "speed")},
      {#'avg'},
      w_time(CarSegStr(), 300, 1, res_idx("CarSegStr", "time"))));


/* SegVol (relation): Relation containing the number of cars currently
in a segment. The relation CurCarSeg is used to determine the cars in
each segment.

SELECT exp_way, dir, seg, COUNT(*) as volume
FROM CurCarSeg
GROUP BY exp_way, dir, seg; */

create function __meta__SegVol() ->
  <Integer exp_way, Integer dir, Integer seg, Integer volume>;

create function __meta__GroupbySegVol() ->
  <Integer exp_way, Integer dir, Integer seg, Integer volume>;

create function SegVol() -> Stream of Window as select
  w_project(
    {res_idx("GroupbySegVol", "exp_way"),
     res_idx("GroupbySegVol", "dir"),
     res_idx("GroupbySegVol", "seg"),
     res_idx("GroupbySegVol", "volume")},
    w_group_by(
      {res_idx("CurCarSeg", "exp_way"),
       res_idx("CurCarSeg", "dir"),
       res_idx("CurCarSeg", "seg")},
      {res_idx("CurCarSeg", "speed")},
      {#'count'},
      CurCarSeg()));


/* SegToll (relation): Relation containing the toll for each
segment. There are no entries in the relation for segments having no
toll. A segment is tolled only if the average speed of the segment is
less than 40, and if it is not affected by an accident (represented
here by the relation AccAffectedSeg which is specified later). If a
segment is tolled, its toll is basetoll * (#cars - 150) * (#cars -
150).

SELECT S.exp_way, S.dir, S.seg, basetoll*(V.volume-150)*(V.volume-150)
FROM SegAvgSpeed as S, SegVol as V
WHERE S.exp_way = V.exp_way and S.dir = V.dir and S.seg = V.seg
      and S.speed < 40
      and (S.exp_way, S.dir, S.seg) NOT IN (AccAffectedSeg);

Pity the lone car of a segment traveling at less than 40! */

create function __meta__SegToll() ->
  <Integer time, Integer exp_way, Integer dir, Integer seg, Integer toll>;

create function __join__SegToll(Vector speed, Vector vol, Vector acc) -> Boolean as
  speed[res_idx("SegAvgSpeed", "exp_way")] = vol[res_idx("SegVol", "exp_way")] and
  speed[res_idx("SegAvgSpeed", "dir")] = vol[res_idx("SegVol", "dir")] and
  speed[res_idx("SegAvgSpeed", "seg")] = vol[res_idx("SegVol", "seg")] and
  speed[res_idx("SegAvgSpeed", "exp_way")] < 40 and True !=
  (speed[res_idx("SegAvgSpeed", "exp_way")] = acc[res_idx("AccAffectedSeg", "exp_way")] and
   speed[res_idx("SegAvgSpeed", "dir")] = acc[res_idx("AccAffectedSeg", "dir")] and
   speed[res_idx("SegAvgSpeed", "seg")] = acc[res_idx("AccAffectedSeg", "seg")]);

create function __toll__SegToll(Vector of Vector row) -> Number as
  666 * (cast(vol[res_idx("SegVol", "volume")] as Number)-150) * (cast(vol[res_idx("SegVol", "vol

create function SegToll() -> Stream of Window as select
  w_project(
    {join_res_idx({"SegAvgSpeed", "SegVol"}, "SegAvgSpeed", "exp_way"),
     join_res_idx({"SegAvgSpeed", "SegVol"}, "SegAvgSpeed", "dir"),
     join_res_idx({"SegAvgSpeed", "SegVol"}, "SegAvgSpeed", "seg"),
     #'__toll__SegToll'},
    w_join(
      {SegAvgSpeed(),
       SegVol(),
       AccAffectedSeg()},
      #'__join__SegToll'));


/*********
 *********
 ********* ACCIDENT DETECTION AND NOTIFICATION
 *********
 *********/

 /* AccCars (relation): Relation containing cars currently involved in
 an accident, and the position of the accident. Note that AVG(x-pos)
 below is just a hack to get the common location of 4 identical
 location reports of a car involved in an accident.

SELECT car_id, AVG(x-pos) AS acc_loc
FROM CarLocStr [PARTITION BY car_id ROWS 4]
GROUP BY car_id
HAVING COUNT DISTINCT (x-pos) == 1; */
```

```
create function __meta__AccCars() ->
  <Integer car_id, Integer acc_loc>;

create function __meta__GroupbyAccCars() ->
  <Integer car_id, Integer acc_loc, Integer pos_count>;

create function __filter__AccCars(Vector row) -> Boolean as
  row[res_idx("GroupbyAccCars", "pos_count")] = 1;

create function AccCars() -> Stream of Window as select
  w_project(
    {res_idx("GroupbyAccCars", "car_id"),
     res_idx("GroupbyAccCars", "acc_loc")},
    w_filter(
      w_group_by(
        {res_idx("CarLocStr", "car_id")},
        {res_idx("CarLocStr", "x_pos"),
         res_idx("CarLocStr", "x_pos")},
        {#'avg',
         #'count_distinct'},
        w_partition(CarLocStr(), {res_idx("CarLocStr", "car_id")}, 4, 1,
                    res_idx("CarLocStr", "time"))),
      #'__filter__AccCars'));


/* AccSeg (relation):  Relation containing the set of segments involved in an accident. This rela
CurCarSeg  relation with  AccCars  relation.

SELECT DISTINCT exp_way, dir, seg, acc_loc
FROM CurCarSeg, AccCars
WHERE CurCarSeg.car_id = AccCars.car_id; */

create function __meta__AccSeg() ->
  <Integer time, Integer exp_way, Integer dir, Integer seg, Integer acc_loc>;

create function __join__AccSeg(Vector car, Vector acc_car) -> Boolean as
  car[res_idx("CurCarSeg", "car_id")] = acc_car[res_idx("AccCars", "car_id")];

create function AccSeg() -> Stream of Window as select
  w_distinct(
    {res_idx("AccSeg", "exp_way"),
     res_idx("AccSeg", "dir"),
     res_idx("AccSeg", "seg"),
     res_idx("AccSeg", "acc_loc")},
    w_project(
      {join_res_idx({"CurCarSeg", "AccCars"}, "time"),
       join_res_idx({"CurCarSeg", "AccCars"}, "exp_way"),
       join_res_idx({"CurCarSeg", "AccCars"}, "dir"),
       join_res_idx({"CurCarSeg", "AccCars"}, "seg"),
       join_res_idx({"CurCarSeg", "AccCars"}, "acc_loc")},
      w_join(
        {CurCarSeg(),
         AccCars()},
        #'__join__AccSeg')));


/* AccNotifyStr (stream): Output stream notifying an accident to cars
currently in the upstream 5 segments from the accident segment. The
ISTREAM operator streams a new accident being inserted into AccSeg
relation; a new accident tuple is joined with CurCarSeg relation to
determine cars in the upstream 5 segments.
```

```
SELECT RSTREAM (car_id, acc_loc)
FROM (ISTREAM (AccSeg)) [NOW] AS A, CurCarSeg as S
WHERE (A.exp_way = S.exp_way and A.dir = EAST and S.dir = EAST and
      S.seg < A.seg and S.seg > A.seg - 5) OR
      (A.exp_way = S.exp_way and A.dir = WEST and S.dir = WEST and
      S.seg > A.seg and S.seg < A.seg + 5); */

create function __meta__AccNotifyStr() ->
  <Integer time, Integer car_id, Integer acc_loc>;

create function __join__AccNotifyStr(Vector acc, Vector car) -> Boolean as
  (acc[res_idx("AccSeg", "exp_way")] = car[res_idx("CurCarSeg", "exp_way")] and
   acc[res_idx("AccSeg", "dir")] = "EAST" and
   car[res_idx("CurCarSeg", "dir")] = "EAST" and
   car[res_idx("CurCarSeg", "seg")] < acc[res_idx("AccSeg", "seg")] and
   car[res_idx("CurCarSeg", "seg")] > cast(acc[res_idx("AccSeg", "seg")] as Number) - 5) or
  (acc[res_idx("AccSeg", "exp_way")] = car[res_idx("CurCarSeg", "exp_way")] and
   acc[res_idx("AccSeg", "dir")] = "WEST" and
   car[res_idx("CurCarSeg", "dir")] = "WEST" and
   car[res_idx("CurCarSeg", "seg")] > acc[res_idx("AccSeg", "seg")] and
   car[res_idx("CurCarSeg", "seg")] < cast(acc[res_idx("AccSeg", "seg")] as Number) + 5);


create function AccNotifyStr() -> Stream of Vector as select
  rstream(w_project(
    {join_res_idx({"AccSeg", "CurCarSeg"}, "time"),
     join_res_idx({"AccSeg", "CurCarSeg"}, "car_id"),
     join_res_idx({"AccSeg", "CurCarSeg"}, "acc_loc")},
    w_join(
      {w_now(istream(AccSeg()), res_idx("AccSeg", "time")),
       CurCarSeg()},
      #'__join__AccNotifyStr')));


/* AccAffectedSeg (relation): Relation of segments not tolled due to
accidents (see SegToll relation). The 10 upstream segments of an
accident segment are not tolled until 20 minutes after an accident is
cleared. For simplicity, the CQL specification of this relation
assumes a fixed relation AllSeg containing the set of all all segments
in all the freeways. The relation AccAffectedSeg is specified below as
a union of two relations---the relation containing 10 upstream
segments of segments currently having an uncleared accident, and the
relation containing 10 upstream segments of segments that had an
accident cleared within the last 20 minutes.

SELECT A.exp_way, A.dir, A.seg
FROM AllSeg AS A, AccSeg AS S
WHERE (A.exp_way = S.exp_way AND A.dir = EAST AND S.dir = EAST AND
       A.seg < S.seg AND A.seg > S.seg - 10) OR
      (A.exp_way = S.exp_way AND A.dir = WEST AND S.dir = WEST AND
       A.seg > S.seg AND A.seg < S.seg + 10)

UNION

SELECT A.exp_way, A.dir, A.seg
FROM AllSeg AS A, DSTREAM ( AccSeg )[RANGE 20 MINUTES] AS S
WHERE (A.exp_way = S.exp_way AND A.dir = EAST AND S.dir = EAST AND
       A.seg < S.seg AND A.seg > S.seg - 10) OR
      (A.exp_way = S.exp_way AND A.dir = WEST AND S.dir = WEST AND
       A.seg > S.seg AND A.seg < S.seg + 10); */
```

```
create function __meta__AccAffectedSeg() ->
  <Integer time, Integer exp_way, Integer dir, Integer seg>;

create function __join_amosfn__AccAffectedSeg(Vector accseg_row) -> Bag of Vector as
  select allseg_row
  from Vector allseg_row
  where allseg_row in streamof(AllSeg()) and
    ((allseg_row[res_idx("AllSeg", "exp_way")] = accseg_row[res_idx("AccSeg", "exp_way")] and
      allseg_row[res_idx("AllSeg", "dir")] = "EAST" and
      accseg_row[res_idx("AccSeg", "dir")] = "EAST" and
      accseg_row[res_idx("AllSeg", "seg")] < allseg_row[res_idx("AccSeg", "seg")] and
      accseg_row[res_idx("AllSeg", "seg")] > cast(allseg_row[res_idx("AccSeg", "seg")] as Number)
     (allseg_row[res_idx("AllSeg", "exp_way")] = accseg_row[res_idx("AccSeg", "exp_way")] and
      allseg_row[res_idx("AllSeg", "dir")] = "WEST" and
      accseg_row[res_idx("AccSeg", "dir")] = "WEST" and
      accseg_row[res_idx("AllSeg", "seg")] > allseg_row[res_idx("AccSeg", "seg")] and
      accseg_row[res_idx("AllSeg", "seg")] < cast(allseg_row[res_idx("AccSeg", "seg")] as Number)

create function AccAffectedSeg() -> Stream of Window as select
  w_union({
    w_project(
      {join_res_idx({"AccSeg", "AllSeg"}, "AccSeg", "time"),
       join_res_idx({"AccSeg", "AllSeg"}, "AllSeg", "exp_way"),
       join_res_idx({"AccSeg", "AllSeg"}, "AllSeg", "dir"),
       join_res_idx({"AccSeg", "AllSeg"}, "AllSeg", "seg")},
      w_join_amosfn(
        AccSeg(),
        #'__join_amosfn__AccAffectedSeg')),
    w_project(
      {join_res_idx({"AccSeg", "AllSeg"}, "AccSeg", "time"),
       join_res_idx({"AccSeg", "AllSeg"}, "AllSeg", "exp_way"),
       join_res_idx({"AccSeg", "AllSeg"}, "AllSeg", "dir"),
       join_res_idx({"AccSeg", "AllSeg"}, "AllSeg", "seg")},
      w_join_amosfn(
       w_time(dstream(AccSeg()), 1200, 1, res_idx("AccSeg", "time")),
       #'__join_amosfn__AccAffectedSeg'))});


/*********
 *********
 ********* NEGATIVE TOLL GENERATION
 *********
 *********/

/* CarExitStr (stream): Stream of events of cars exiting from the
freeways. A car has exited if it is no longer "active" (i.e., it has
not reported its location in the last 30 seconds). The exiting of cars
is determined by applying the DSTREAM operator on the CurActiveCars
relation. The relation CurActiveCars does not store the latest segment
of the car; this is determined by using the last location report of
the car in CarSegStr.

SELECT RSTREAM (S.car_id, S.exp_way, S.dir, S.seg)
FROM (DSTREAM (CurActiveCars)) [NOW] AS A,
     CarSegStr [PARTITION BY car_id ROWS 1] AS S
WHERE A.car_id = S.car_id; */

create function __meta__CarExitStr() ->
  <Integer time, Integer car_id, Integer exp_way, Integer dir, Integer seg>;

create function __join__CarExitStr(Vector car, Vector loc) -> Boolean as
  car[res_idx("CurActiveCars", "car_id")] = loc[res_idx("CarSegStr", "car_id")];
```

```
create function CarExitStr() -> Stream of Vector as select
  rstream(
    w_project(
      {join_res_idx({"CurActiveCars", "CarSegStr"}, "CurActiveCars", "time"),
       join_res_idx({"CurActiveCars", "CarSegStr"}, "CarSegStr", "car_id"),
       join_res_idx({"CurActiveCars", "CarSegStr"}, "CarSegStr", "exp_way"),
       join_res_idx({"CurActiveCars", "CarSegStr"}, "CarSegStr", "dir"),
       join_res_idx({"CurActiveCars", "CarSegStr"}, "CarSegStr", "seg")},
      w_join(
        {w_now(dstream(CurActiveCars()), res_idx("CurActiveCars", "time")),
         w_partition(CarSegStr(), {res_idx("CarSegStr", "car_id")}, 1, 1,
                     res_idx("CarSegStr", "time"))},
        #'__join__CarExitStr')));


/* NegTollStr (stream): Stream of negative tolls to cars exiting on a
segment. The amount of the negative toll for an exiting car is equal
to the latest charged toll (TollStr) for the car.

SELECT RSTREAM (E.car_id, T.toll)
FROM CarExitStr [NOW] AS E, TollStr [PARTITION BY car_id ROWS 1] AS T
WHERE E.car_id = T.car_id; */

create function __meta__NegTollStr() ->
  <Integer time, Integer car_id, Integer exp_way, Integer dir, Integer seg>;

create function __join__NegTollStr(Vector car, Vector toll) -> Boolean as
  car[res_idx("CarExitStr", "car_id")] = toll[res_idx("TollStr", "car_id")];

create function NegTollStr() -> Stream of Vector as select
  rstream(
    w_project(
      {join_res_idx({"CarExitStr", "TollStr"}, "CarExitStr", "time"),
       join_res_idx({"CarExitStr", "TollStr"}, "CarExitStr", "car_id"),
       join_res_idx({"CarExitStr", "TollStr"}, "TollStr", "toll")},
      w_join(
        {w_now(CarExitStr(), res_idx("CarExitStr", "time")),
         w_partition(TollStr(), {res_idx("TollStr", "car_id")}, 1, 1,
                     res_idx("TollStr", "time"))},
        #'__join__NegTollStr')));

/* NegAccTollStr (stream): Stream of negative tolls given to cars
exiting less than five segments upstream from an accident segment
(relation AccSeg).

SELECT RSTREAM (car_id, X)
FROM CarExitStr [NOW] as E, AccSeg as A
WHERE (E.exp_way = A.exp_way AND E.dir = EAST AND A.dir = EAST and
       E.seg < A.seg AND E.seg > A.seg - 5) OR
      (E.exp_way = A.exp_way AND E.dir = WEST AND A.dir = WEST and
       E.seg > A.seg AND E.seg < A.seg + 5); */

create function __meta__NegAccTollStr() ->
  <Integer time, Integer car_id, Integer neg_toll>;

create function __join__NegAccTollStr(Vector car, Vector acc) -> Boolean as
  (car[res_idx("CarExitStr", "exp_way")] = acc[res_idx("AccSeg", "exp_way")] and
   car[res_idx("CarExitStr", "dir")] = "EAST" and
   acc[res_idx("AccSeg", "dir")] = "EAST" and
   acc[res_idx("AccSeg", "seg")] < car[res_idx("CarExitStr", "seg")] and
   acc[res_idx("AccSeg", "seg")] > cast(car[res_idx("CarExitStr", "seg")] as Number) - 5) or
  (car[res_idx("CarExitStr", "exp_way")] = acc[res_idx("AccSeg", "exp_way")] and
   car[res_idx("CarExitStr", "dir")] = "WEST" and
   acc[res_idx("AccSeg", "dir")] = "WEST" and
   acc[res_idx("AccSeg", "seg")] > car[res_idx("CarExitStr", "seg")] and
   acc[res_idx("AccSeg", "seg")] < cast(car[res_idx("CarExitStr", "seg")] as Number) + 5);

create function __neg_toll__NegAccTollStr(Vector row) -> Number as
  42;

create function NegAccTollStr() -> Stream of Vector as select
  rstream(
    w_project(
      {join_res_idx({"CarExitStr", "AccSeg"}, "CarExitStr", "time"),
       join_res_idx({"CarExitStr", "AccSeg"}, "car_id"),
       #'__neg_toll__NegAccTollStr'},
      w_join(
        {w_now(CarExitStr(), res_idx("CarExitStr", "time")),
         AccSeg()},
        #'__join__NegAccTollStr')));

/*********
 *********
 ********* ACCOUNTING
 *********
 *********/

/* AccTransStr (stream): The union of all the transaction of a car
account - the "historical" credit stream (CreditStr), the toll stream
(TollStr), the negative toll stream (NegTollStr), and the negative
toll stream due to accidents (NegAccTollStr).

SELECT *
FROM CreditStr

UNION ALL

SELECT car_id, toll AS credit
FROM NegTollStr

UNION ALL

SELECT car_id, toll AS credit
FROM NegAccTollStr

UNION ALL

SELECT car_id, -1 * toll AS credit
FROM TollStr; */

create function __filtered__NegTollStr() -> Stream of Vector as select streamof(
  select
    {v[res_idx("NegTollStr", "time")],
     v[res_idx("NegTollStr", "car_id")],
     v[res_idx("NegTollStr", "toll")]}
  from Vector v
  where v in NegTollStr());

create function __filtered__NegAccTollStr() -> Stream of Vector as select streamof(
  select
    {v[res_idx("NegAccTollStr", "time")],
     v[res_idx("NegAccTollStr", "car_id")],
     v[res_idx("NegAccTollStr", "toll")]}
  from Vector v
  where v in NegAccTollStr());

create function __filtered__TollStr() -> Stream of Vector as select streamof(
  select
    {v[res_idx("TollStr", "time")],
     v[res_idx("TollStr", "car_id")],
     -1 * cast(v[res_idx("TollStr", "toll")] as Number)}
  from Vector v
  where v in TollStr());

create function __meta__AccTransStr() ->
  <Integer time, Integer car_id, Integer credit>;

create function AccTransStr() -> Stream of Vector as select
  s_union_all({
    CreditStr(),
    __filtered__NegTollStr(),
    __filtered__NegAccTollStr(),
    __filtered__TollStr()});

/* AccBal (relation): Relation corresponding to the current account
balance of each car.

SELECT car_id, SUM(credit) AS balance
FROM AccTransStr
GROUP BY car_id; */

create function __meta__AccBal() ->
  <Integer car_id, Integer balance>;

create function __meta__GroupbyAccBal() ->
  <Integer car_id, Integer balance>;

create function AccBal() -> Stream of Window as select
  w_project(
    {res_idx("GroupbyAccBal", "car_id"),
     res_idx("GroupbyAccBal", "balance")},
    w_group_by(
      {res_idx("AccTransStr", "car_id")},
      {res_idx("AccTransStr", "credit")},
      {#'sum'},
      w_now(AccTransStr(), res_idx("AccTransStr", "time"))));



/*********
 *********
 ********* ADHOC QUERY ANSWERING
 *********
 *********/

/* AccBalOutStr (stream):  Output stream of account balance queries.

SELECT RSTREAM(query_id, B.car_id, B.balance)
FROM AccBalQueryStr [NOW] AS Q, AccBal AS B
WHERE Q.car_id = B.car_id; */

create function __meta__AccBalOutStr() ->
  <Integer time, Integer query_id, Integer car_id, Integer balance>;

create function __join__AccBalOutStr(Vector query, Vector bal) -> Boolean as
```

```
  query[res_idx("AccBalQueryStr", "car_id")] = bal[res_idx("AccBal", "car_id")];

create function AccBalOutStr() -> Stream of Vector as select
  rstream(
    w_project(
      {join_res_idx({"AccBalQueryStr", "AccBal"}, "AccBalQueryStr", "time"),
       join_res_idx({"AccBalQueryStr", "AccBal"}, "query_id"),
       join_res_idx({"AccBalQueryStr", "AccBal"}, "car_id"),
       join_res_idx({"AccBalQueryStr", "AccBal"}, "balance")},
      w_join(
        {w_now(AccBalQueryStr(), res_idx("AccBalQueryStr", "time")),
         AccBal()},
        #'__join__AccBalOutStr')));

/* ExpOutStr (stream):  Output stream of current-day-expenditure adhoc queries.

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;

Note that this query is incomplete, i.e., the alias E is used but never defined.
Furthermore, the aggregate function sum is applied on AccTranStr.credit but the
group by operator is never used. We will rewrite the query:

ExpToday (relation): Relation of todays expenditure for each car.

SELECT car_id, SUM(credit) as expenditure
FROM AccTransStr [RANGE 24 HOURS SLIDE 24 HOURS]
GROUP BY car_id;

ExpOutStr (stream): Output stream of current-day-expenditure adhoc queries.

SELECT RSTREAM(time, query_id, car_id, expenditure)
FROM ExpQueryStr [NOW] as Q, ExpToday as T
WHERE Q.car_id = T.car_id; */

create function __meta__ExpToday() ->
  <Integer car_id, Integer expenditure>;

create function __meta__GroupbyExpToday() ->
  <Integer car_id, Integer expenditure>;

create function ExpToday() -> Stream of Window as select
  w_project(
    {res_idx("GroupbyExpToday", "car_id"),
     res_idx("GroupbyExpToday", "expenditure")},
    w_group_by(
      {res_idx("AccTransStr", "car_id")},
      {res_idx("AccTransStr", "credit")},
      {#'sum'},
      w_time(AccTransStr(), 24*60*60, 24*60*60, res_idx("AccTransStr", "time"))));

create function __meta__ExpOutStr() ->
  <Integer time, Integer query_id, Integer car_id, Integer expenditure>;

create function __join__ExpOutStr(Vector query, Vector trans) -> Boolean as
  query[res_idx("ExpQueryStr", "car_id")] = trans[res_idx("AccTransStr", "car_id")];

create function ExpOutStr() -> Stream of Vector as select
  rstream(
    w_project(
      {join_res_idx({"ExpQueryStr", "ExpToday"}, "time"),
       join_res_idx({"ExpQueryStr", "ExpToday"}, "query_id"),
       join_res_idx({"ExpQueryStr", "ExpToday"}, "car_id"),
       join_res_idx({"ExpQueryStr", "ExpToday"}, "expenditure")},
      w_join(
        {w_now(ExpQueryStr(), res_idx("ExpQueryStr", "time")),
         ExpToday()},
        #'__join__ExpOutStr')));
```

B Source Code

The FCQL source code is available at http://user.it.uu.se/ roka4241/fcql.

B.1 Window Datatype

Continuous relations are represented using a new window datatype which is implemented in C and used by most FCQL operators.

/scsq/include/swin.h Stream window datatype header.

/scsq/C/swin.c Stream window datatype source.

B.2 FCQL Operators

FCQL operators are implemented in ALisp and found in */scsq/cql/operators/*.

/scsq/cql/operators/window.lsp Relation-to-relation operators.

/scsq/cql/operators/aggregate.lsp Continuous relation aggregation.

/scsq/cql/operators/stream.lsp Relation-to-stream operators.

/scsq/cql/operators/join.lsp Stream and continuous relation joining.

B.3 Common Functionality

/scsq/cql/lsp/base.lsp Common FCQL functionality.

B.4 Regressions tests

The correctness of the FCQL system is verified using a series of regressions tests which can be found in */scql/cql/regress/*. These tests make use of several test data files available in */scql/cql/data/*.

/scsq/cql/regress/window.osql Relation-to-relation operator tests.

/scsq/cql/regress/aggregate.osql Continuous relation aggregation tests.

/scsq/cql/regress/stream.osql Relation-to-stream operator tests.

/scsq/cql/regress/join.osql Stream and continuous relation joining tests.

/scsq/cql/regress/base.osql Common FCQL functionality tests.