Towards service-oriented continuous queries in pervasive systems

Yann Gripay* — **Frédérique Laforest*** — **Jean-Marc Petit***

* *Université de Lyon, INSA-Lyon, LIRIS – UMR 5205 CNRS*
7 avenue Jean Capelle, F-69621 Villeurbanne, France
{yann.gripay,frederique.laforest,jean-marc.petit}@liris.cnrs.fr

ABSTRACT. *Pervasive information systems give an overview of what digital environments should look like in the future. From a data-centric point of view, traditional databases have to be used alongside with non-conventional data sources like data streams, services and events. In this paper, we tackle the definition of continuous queries combining standard relations, data streams and services in a declarative language extending SQL. We first define virtual tables with binding patterns as a way to get a unified view of the pervasive environment. Relations, data streams and services can be homogeneously queried using a SQL-like language, on top of which query optimization can be performed. We also introduce a new clause defining the optimizing criteria to dynamically choose the best way to handle each event.*

RÉSUMÉ. *Les systèmes d'information pervasifs montrent la tendance sur ce que seront les environnements informatiques de demain. D'un point de vue centré données, les bases de données classiques doivent cohabiter avec des sources de données non-conventionnelles comme les flux de données, les services et les évènements. Dans cet article, nous abordons la définition de requêtes continues combinant les relations classiques, les flux de données et les services dans un langage déclaratif étendant SQL. Nous définissons tout d'abord les tables virtuelles avec des binding patterns afin d'obtenir une vue unifiée de l'environnement pervasif. Relations, flux de données et services peuvent être utilisés de manière homogène dans des requêtes exprimées dans un langage à la SQL, sur lesquelles une optimisation de requête peut être effectuée. Nous introduisons également une nouvelle clause définissant les critères d'optimisation permettant de choisir dynamiquement le moyen optimum de traiter chaque évènement.*

KEYWORDS: *Continuous Queries, Non-Conventional Data Sources, Query Optimization, Data Streams, Services, Pervasive Systems*

MOTS-CLÉS : *Requêtes continues, Sources de données non-conventionnelles, Optimisation de Requêtes, Flux de données, Services, Systèmes pervasifs*

RSTI - ISI – 13/2008. Modèles et langages pour les bases de données, pages 33 à 57

1. Introduction

Pervasive information systems give an overview of what digital environments should look like in the future. Information systems tend to be more and more decentralized and autonomous, at the infrastructure level as well as at the data and process level. On the one hand, personal computers and other handheld devices are now democratized and take a large part of information systems. On the other hand, data sources may be distributed over large area through networks that range from a worldwide network like the Internet to local peer-to-peer connections like for sensors.

Even data tend to change their form to handle information dynamicity. The relational paradigm has been widely adopted in DataBase Management Systems (DBMS) for many years, but other forms of data sources are now emerging, mainly as data streams and services.

Queries in traditional DBMS are "snapshot queries" expressed in SQL: a query is evaluated with the current state of the database, and the result is a static relational table. The "snapshot" term expresses that the result represents only the state of the database at the moment of the query, and is never updated. With dynamic data sources, "snapshot queries" may be not sufficient as it would be computation-expensive to periodically execute them and obtain up-to-date results.

Data streams open new opportunities to view and manage dynamic systems, such as sensor networks. The concept of queries that last in time, called *continuous queries* (Chen *et al.*, 2000), allows to define queries whose results are continuously updated as data "flow" in the data streams. Data Stream Management Systems (DSMS) have been studied in many works (Abadi *et al.*, 2005; Arasu *et al.*, 2003; Chandrasekaran *et al.*, 2003; Cherniack *et al.*, 2003; Franklin *et al.*, 2005b; Tian *et al.*, 2003; Yao *et al.*, 2003).

With the development of autonomous devices and location-dependent functionalities, information systems tend to become what Mark Weiser (Weiser, 1991) called ubiquitous systems, or pervasive systems. Pervasive systems (Becker *et al.*, 2004; Brumitt *et al.*, 2000; Estrin *et al.*, 2002; Garlan *et al.*, 2002; Grimm *et al.*, 2004) are distributed systems of devices able to communicate with others through network links. They offer to users access to devices and control over their environment through various types of interfaces.

The abstraction of device functionalities allows the system to automate some of the possible interactions between heterogeneous devices, in order to facilitate the use of the whole system. Such device functionalities are often represented by services. As devices may be sensors or effectors, services may represent some interactions with the physical environment, like taking a photo from a camera or displaying a picture on a screen. These interactions bridge the gap between the computing environment and the user environment, and can be managed by the pervasive system through such services.

In this paper, we will consider a way to use and compose services with the notion of *service-oriented* queries. From a data-centric point of view, traditional databases have

to be used alongside with non-conventional data sources like data streams, services and events to deal with new properties such as dynamicity, autonomy and decentralization. Query languages and processing techniques need to be adapted to those data sources. Data management systems tend to evolve from DBMS or DSMS to a more general concept of DataSpace Support Plateform (DSSP) (Franklin *et al.*, 2005a). A DSSP is intended to deal with "large amount of interrelated but disparately managed data". In this context, the definition of continuous queries combining standard relations, data streams and services in a declarative language extending SQL is clearly an ambitious and motivating goal. We begin by illustrating the problem with an example that will be used as a running example throughout this paper.

1.1. *Motivating example*

The motivating example is inspired by the night surveillance scenario presented in Aorta (Xue *et al.*, 2005). It illustrates the need for the integration of services from a dynamic environment in a declarative query language and for associated optimization techniques.

The night surveillance scenario considers a room containing motion sensors and network cameras. The surveillance consists of handling events from motion sensors to trigger a photo of the location of the involved sensor and to send it to the administrators via their cell phones. The cameras need to pan/tilt/zoom to focus on a given location (if achievable) before actually taking the photo. This configuration phase is costly in term of response time to an event and depends on the dynamic state of the device (current head position of the camera), so a cost-based evaluation of the optimal device is needed.

In order to express this behavior in a declarative way, the environment can be described using data schemas for the entities and the events, and functions for the interactions with the devices. Then, a query language similar to SQL can express the specified behavior in terms of joins, selections and functions. Query optimization techniques can be applied to optimize the entire process.

In Aorta (Xue *et al.*, 2005), this environment is modeled using three data sources: a relation containing phone numbers of administrators, a data stream for sensor events (indicating its current location and its horizontal acceleration value 'accel_x'), and a "virtual device table" for cameras. Three functions are also needed for the scenario: taking a photo, sending a photo to a cell phone, and checking that a camera is able to take a photo of a location.

The continuous query for the night surveillance scenario is given in AortaSQL (Xue *et al.*, 2005) in Table 1: an *Action Query* called "night_surveillance" is active from midnight to 6:00 am every day (cf. START and STOP clauses).

Table 1. *Query in AortaSQL for the night surveillance scenario from Aorta*

```
CREATE AQ night_surveillance AS
  SELECT sendphoto( p.number, photo(c.ip_address,s.location,"photos/admin") )
  FROM   sensor s, camera c, phone p
  WHERE  s.accel_x > 500
    AND  coverage( c.id, s.location )
    AND  p.owner = "admin"
  START  atTime(0,0,0) -- 00:00:00
  STOP   atTime(6,0,0) -- 06:00:00
```

Despite the interest of Aorta, the following observations can be made:

1) at the query language definition level, no clear distinction is made between event management and stream management. For example, in the above scenario, an event is represented as a tuple in the "sensor" data stream, but is however still handled as an event: it triggers a single interaction with a device (taking one photo) and may not be duplicated due to a join with a relation or another stream. This semantics is not compatible with other DSMS like in (Arasu *et al.*, 2003; Yao *et al.*, 2003; Franklin *et al.*, 2005b; Chandrasekaran *et al.*, 2003);

2) the optimizing criteria are implicit: in the above scenario, the goal of the query is to choose the camera with the least estimated response time for each event, and cannot be declaratively modified to choose another criterion like, for example, the photo quality;

3) at the query processing level, logical and physical steps seem to be merge in a single step. This choice limits the opportunities for query optimization techniques;

4) only limited support is provided for continuous query processing. Specific operators for streams, like windows over streams (Ding *et al.*, 2004) or relation-to-stream operators (Arasu *et al.*, 2003), are not tackled, as well as joining several streams, relations and virtual device tables.

Expressing queries such as the night surveillance scenario requires a framework that remains compatible with standard continuous query processing, allowing to reuse the query optimization techniques of DSMS, and that integrates the notion of interaction with devices like in Aorta.

1.2. *Evolution of continuous queries*

In this paper, we present an ongoing effort to develop a framework for *Service-oriented Continuous Queries* (SoCQs), whose aim is to integrate services, *i.e.* distributed functionalities, in continuous queries over data streams. SoCQs allow the definition of queries combining standard relations, data streams and services using a homogeneous representation, in a declarative language extending SQL.

The first requirement to achieve this ambitious goal is to define a common framework to deal with non-conventional data sources. Relations and data streams can share the same representation as time-varying multisets of tuples like in (Arasu *et al.*, 2003).

We propose to represent sets of similar services as virtual tables containing a tuple per service and associated with one or more binding patterns (Florescu *et al.*, 1999; Goldman *et al.*, 2000; Srivastava *et al.*, 2006) indicating which virtual attributes correspond to input and output parameters of the service functions. We keep backward compatibility with standard DBMS as we use standard relations, while extending the power of expression of queries to handle the notion of time. Event flows are represented as data streams, in order to avoid the mismatch between events and standard data tuples.

SoCQs can imply services that are statically bound (Goldman *et al.*, 2000; Srivastava *et al.*, 2006) or dynamically discovered in the pervasive information system, like in (Pigeot *et al.*, 2007). In pervasive environments, those queries can use the services to access to distributed functionalities. The optimal services (at a given time for a given data set) are selected and called during query execution. SoCQs can then express an event management functionality like event filtering and composition, and perform cost-based optimal calls to services. Continuous queries can evolve from data-oriented queries to service-oriented queries.

In this setting, the main contributions of this paper are :

– an extension of SQL to homogeneously express operators over relations, data streams and services, and an associated query processing technique to handle time-variations of data and dynamic calls to services during execution. An additional COLLAPSE clause in the SQL syntax is proposed to define an optimizing criterion over groups of tuples;

– the development of a prototype of a query processor for SoCQs, from which first experimental results over synthetic data are described. The SoCQ processor is inspired by the STREAM prototype (Arasu *et al.*, 2003), a DSMS developed at Stanford University, and allows to show both the power of expression of SoCQs and the capabilities of the query processor.

In Section 2, we situate our problem within the related works. In Section 3, we define a homogeneous representation for non-conventional data sources as virtual tables. We tackle query processing techniques for virtual tables and the COLLAPSE clause in Section 4. We describe our implementation prototype and discuss some experimental results in Section 5. We then conclude and discuss some open issues in Section 6.

2. Related work

2.1. *Data streams*

In modern information systems, some data sources may generate continuous unbounded streams of data elements. For compatibility with the relational model, data streams are commonly modeled as an append-only multiset of timestamped tuples whereas relations are considered as time-varying multisets of tuples (creation, update, deletion) as in (Arasu *et al.*, 2003). This widely adopted model (Abadi *et al.*, 2005; Chandrasekaran *et al.*, 2003; Cherniack *et al.*, 2003; Franklin *et*

al., 2005b; Tian *et al.*, 2003; Yao *et al.*, 2003) allows to manage structured data streams along with relations.

Time is an important notion for data streams. Tuples have an order in the stream, which is often supposed to be the order of arrival, and are timestamped. Timestamps are also supposed to reference a shared system clock, otherwise a synchronization mechanism is required (Barga *et al.*, 2006).

2.2. *Costly data sources*

Some data sources or function evaluations may be slow, like web services or sensed attributes. Introducing asynchronous calls to data sources and synchronization operators in query execution plans, like in (Goldman *et al.*, 2000), allows to process incomplete tuples until their costly attributes are required, which gives time to complete the asynchronous calls and fill in the missing attribute values. (Xue *et al.*, 2005) introduces a selection among possible candidates (devices offering the same service) based on their current state, to choose the optimal way of evaluating a function, here interacting with a device in a pervasive environment. Furthermore, group optimization allows to optimally distribute simultaneous function evaluations among the possible candidates.

2.3. *Continuous queries*

Continuous queries over data streams are based on the relational paradigm. Standard query operators on relations (Select, Project, Join, Aggregate…) are then used, but their semantics may be unclear or ambiguous. (Arasu *et al.*, 2003) identifies three categories of operators to work with streams and relations: relation-to-relation (standard operators), relation-to-stream, and stream-to-relation. Stream-to-stream operators are absent because they can be composed from other operators. A continuous query is a tree of operators with streams and/or relations as input, and a stream or a relation as output. Some systems (Yao *et al.*, 2003; Abadi *et al.*, 2005; Franklin *et al.*, 2005b) do not express the difference between operator categories, and work, in their semantics, only with data streams.

Unbounded tuple streams potentially require unbounded memory space in order to be joined, as every tuple should be stored to be compared with every tuple from the other stream. Tuple sets should then be bounded: a window defines a bounded subset of tuples from a stream (it is the only stream-to-relation operator in (Arasu *et al.*, 2003)), based on time or on the number of tuples. Sliding windows (Arasu *et al.*, 2003; Ding *et al.*, 2004) have a fixed size and continuously move forward (e.g. the last 100 tuples, tuples within the last 5 minutes). Hopping windows (Yao *et al.*, 2003) have a fixed size and move by hop, defining a range of interval (e.g. 5-minute window every 5 minutes). In (Chandrasekaran *et al.*, 2003), windows can be defined in a flexible way: the window upper and lower bound are defined separately (fixed, sliding or

hopping), allowing various type of windows. (Arasu *et al.*, 2003) also defines a partitioned window as the union of windows over a partitioned stream based on attribute values (e.g. the last 5 tuples for every different ID). With windows, join operators handle bounded sets of tuples and traditional techniques can be applied. Although the output is intuitively thought as a stream, join operators are seen in (Arasu *et al.*, 2003) as relation-to-relation operators: the output is a time-varying relation.

Continuous queries can be expressed in a declarative language. Most of the articles (Arasu *et al.*, 2003; Chandrasekaran *et al.*, 2003; Franklin *et al.*, 2005b; Yao *et al.*, 2003) propose an extension of SQL in order to work with both relational databases and data streams. Some articles (Chen *et al.*, 2000) tackle continuous querying over distributed XML data sets and propose an extension of XML-QL. Others (Abadi *et al.*, 2005) are based on a box representation of operators, expressing queries as a flow of tuples. However, when working with the data stream semantics mixed with the relational paradigm, SQL tends to be widely adopted as a base for query language extensions. Data streams are then represented using a relation schema.

The long-running nature of continuous queries changes the definition of execution plans in order to handle data streams. One method is the construction of a global execution plan, like in (Abadi *et al.*, 2005; Arasu *et al.*, 2003; Franklin *et al.*, 2005b; Yao *et al.*, 2003), which is an extension of a standard execution plan where input and output of operators are queues of tuples instead of relations. As several queries may be running simultaneously, the system can share common operators among the different queries. Another method (Chandrasekaran *et al.*, 2003) is to dynamically distribute tuples to one of their next possible operators (called Eddies), each tuple creating its own execution plan depending on the dynamic state of the system.

3. Dealing with non-conventional data sources

Non-conventional data sources are data sources that cannot be represented as tuples in standard relations, like in conventional databases. The transactional paradigm cannot be directly applied to a data management system that handles dynamic sources like data streams, or dynamically discovered services.

For the purpose of integrating non-conventional data sources in an augmented DBMS, we propose a homogeneous representation of relations, data streams and services through the notion of tables and virtual tables. We keep the presentation rather informal, the basic notions being simple.

3.1. *Relations and data streams*

A *relation schema* is a name associated with a set of attributes. Each *attribute* has a name and a definition domain of atomic values. A *tuple* over a relation schema is an element of the Cartesian product of its attribute domains. A *relation* over a relation schema is a multiset of tuples. Tuples can be inserted in a relation, and be later deleted

from it. A *stream* can be defined as a relation where tuples cannot be deleted, *i.e.* an append-only multiset of tuples. Tuples inserted in a stream are associated with their insertion date.

The following definition of a table is inspired by the work on data streams in (Arasu *et al.*, 2003) and the associated prototype. As data sources are dynamic, the notion of time needs to be explicit, in contrast with the transactional paradigm. Time is represented as a discrete and ordered domain of *timestamps* (e.g. positive integer values). Two events are simultaneous if they are both associated with the same timestamp.

In order to homogeneously represent relations and streams, we define a *table* over a relation schema as a multiset of tuples associated with their insertion timestamps. In other words, a table represents a relation where each tuple is associated with its insertion timestamp. A table represents a stream if no tuples can be deleted from the table. With this definition, a table can homogeneously represent a relation or a stream.

We consider the *instantaneous relation* (Arasu *et al.*, 2003) of a table at a given timestamp as the multiset of tuples that have been inserted until this timestamp included, and that have not yet been deleted. Note that a tuple can be inserted and deleted simultaneously, *i.e.* at the same timestamp. For a table representing a stream, the number of tuples of its instantaneous relation may only grow, as no tuple can be deleted: a stream is unbounded.

Example 1 (Tables for relations and streams) *Table 2 and Table 3 show two tables representing a relation "phone" and a stream "sensor". The instantaneous relations for both tables are represented at timestamp 25 and at timestamp 30. Note that at timestamp 30, the tuple "Bob" has been deleted from the "phone" table. Note also that several tuples can be inserted simultaneously, like at timestamp 27 in the "sensor" table.*

Table 2. *Schema and two instantaneous relations for the table representing the "phone" relation*

```
TABLE phone( id INTEGER, owner CHAR(10), number CHAR(12))
```

Timestamp @ 25	Timestamp @ 30
(34,"Alice","+3369911XXXX") @ 10	(34,"Alice" ,"+3369911XXXX") @ 10
(25,"Bob" ,"+3369922XXXX") @ 12	(18,"Charlie","+3369933XXXX") @ 26
	(24,"David" ,"+3369944XXXX") @ 28

Table 3. *Schema and two instantaneous relations for the table representing the "sensor" stream*

```
TABLE sensor( id INTEGER, accel_x FLOAT, location BYTE)
```

Timestamp @ 25	Timestamp @ 30
(18, 362.15, 'a') @ 16	(18, 362.15, 'a') @ 16
(65, 569.42, 'e') @ 25	(65, 569.42, 'e') @ 25
	(18, 236.78, 'a') @ 27
	(17, 718.64, 'd') @ 27
	(98, 624.16, 'c') @ 28

3.2. *Services*

A *service* is an external entity (in regard to the query management system) that can compute one or more functions. We define a *service interface* as a group of semantically related functions. A function can have several input parameters (may be none) and several output parameters (at least one). When called with atomic values for its input parameters, a function returns zero, one or several result lines of atomic values, each line containing all output parameters.

Example 2 (Service interface) *Table 4 shows the definition of a service interface providing three functions:* checkCoverage() *that indicates if the service can take a photo of a given location,* checkCost() *that indicates the cost of taking this photo, and* takePhoto() *that actually takes it.*

Table 4. *Example of service interface*

```
SERVICE INTERFACE cameraInterface {
  FUNCTION checkCoverage( target BYTE ) : ( status BOOLEAN )
  FUNCTION checkCost( target BYTE ) : ( status FLOAT )
  FUNCTION takePhoto( target BYTE ) : ( result BLOB )
}
```

To smoothly integrate services in our framework, we propose to use the notion of *binding pattern*. A *binding pattern* models an access pattern to a relational data source as a specification of "which attributes of a relation must be given values when accessing a set of tuples" (Florescu *et al.*, 1999). A relation with binding patterns can represent an external data source with limited access patterns (Florescu *et al.*, 1999) in the context of data integration. It can also represent an interface to an infinite data source like a web site search engine (Goldman *et al.*, 2000), providing a list of URLs corresponding to some given keywords. In a more general way, it can represent a data service, e.g. web services providing data sets, as a virtual relational table like in (Srivastava *et al.*, 2006).

In our framework, we propose to define a *virtual table* as a generalization of our notion of table: its schema can contain *virtual attributes* and is associated with *binding patterns* involving functions from a service interface. A *virtual attribute* is an attribute whose value is set during query execution, *i.e.* is not set when the tuple is retrieved from the data source. A *binding pattern* is a rule that indicates which function from the service interface has to be invoked in order to retrieve the values of some virtual attributes (the output parameters) when values are set for some other virtual attributes (the input parameters).

Example 3 (Binding patterns) *Table 5 shows the definition of a virtual table "camera" and its associated binding patterns using the service interface* cameraInterface *given in Example 2. The virtual table schema contains one non-virtual attribute "id" and four virtual attributes. When a value is given for the virtual attribute "location", the three binding patterns can be invoked if needed to independently retrieve the values of the other virtual attributes "coverage", "cost" and "photo".*

Table 5. *Schema and binding patterns for the virtual table "camera"*

```
VIRTUAL TABLE camera ( id INTEGER,
                       location BYTE VIRTUAL,
                       coverage BOOLEAN VIRTUAL,
                       cost FLOAT VIRTUAL,
                       photo BLOB VIRTUAL )

BINDING PATTERNS FOR camera USING cameraInterface {
  FUNCTION checkCoverage( location ) : ( coverage )
  FUNCTION checkCost( location ) : ( cost )
  FUNCTION takePhoto( location ) : ( photo )
}
```

A virtual table, like non-virtual tables, contains tuples. However, as those tuples contains virtual attributes, we refer to them as *virtual tuples*. Each virtual tuple is bound to one service that implements the service interface used by the virtual table. A reference to the bound service is stored in a special type of attribute: a service reference attribute. During query execution, when a binding pattern is invoked for a virtual tuple, the required function is invoked from the service to which this virtual tuple is bound. Like tuples in a table, virtual tuples can be inserted in a virtual table, and deleted from it.

Example 4 (Virtual tuples) *Continuing the previous example, Table 6 shows instantaneous relations for the virtual table "camera",* i.e. *the virtual tuples it contains, at timestamp 25 and 30. Only the non-virtual attribute "id" has a value. The "*" indicates that no value is set for the four virtual attributes "location", "coverage", "cost" and "photo". Each virtual tuple is bound to a service, indicated by the service reference, e.g. "Camera2", "Camera3". Note that the tuple bound to the service "Camera2" at timestamp 25 does no longer belong to the table at timestamp 30, because the service itself is no longer available in the pervasive environment.*

Table 6. *Two instantaneous relations at different timestamps for the virtual table "camera"*

```
Timestamp @ 25
(2, *, *, *, *) # Camera2 @ 12
(3, *, *, *, *) # Camera3 @ 12
(5, *, *, *, *) # Camera5 @ 25

Timestamp @ 30
(3, *, *, *, *) # Camera3 @ 12
(5, *, *, *, *) # Camera5 @ 25
(8, *, *, *, *) # Camera8 @ 27
(6, *, *, *, *) # Camera6 @ 28
```

In other words, a virtual table represents a set of services providing the same functionalities, *i.e.* implementing the same service interface. Tuples can be dynamically inserted and deleted whenever such services are discovered in a pervasive environment. The services can also be manually added by a system developer. An extreme

case is a virtual table containing one and only one *static virtual tuple*, *i.e.* a virtual tuple that cannot be deleted: the virtual table is then a simple interface to one statically bound service, or even one function, as it is used in previous works (Florescu *et al.*, 1999; Goldman *et al.*, 2000; Srivastava *et al.*, 2006). We call such a virtual table, a *static virtual table*, as opposed to the general case, a *dynamic virtual table*.

Example 5 (Environment for the night surveillance) *Using our framework, the environment for the night surveillance scenario (described in the motivating example) can be represented in a homogeneous way with four tables. Along with the "phone" and "sensor" tables defined in Example 1, and the "camera" virtual table defined in Example 3, one more table is required: a static virtual table "sendMMS" representing a function that sends a MMS (Multimedia Message) to a cell phone.*

To end up, virtual tables generalize the notion of table representing a relations or a stream. It can then be thought as a homogeneous representation for all data sources needed in a pervasive environment: relations, streams, static and dynamic virtual tables. Table 7 summarizes the constraints for each type of data sources.

Table 7. *Summary of constraints for each type of data sources*

Type of Data Source	Tuple Insertion	Tuple Deletion	Binding Patterns
Relation	yes	yes	no
Stream	yes	no	no
Static Virtual Table	no	no	yes
Dynamic Virtual Table	yes	yes	yes

System developers can work with a common representation of the different data sources available in their computing environment. More importantly, they can devise their queries involving different types of data sources using a single SQL-like declarative language, without worrying about the particular implementation of the data sources. As such, the way we model the environment is a contribution towards the notion of dataspace (Franklin *et al.*, 2005a).

4. Query processing for SoCQs

SoCQs are continuous queries over tables for relations and data streams, and virtual tables for functions and services. Simple queries could be expressed using a SQL-like declarative language. CQL (Continuous Query Language (Arasu *et al.*, 2003)) provides syntax extensions to SQL in order to handle the specificities of data streams and to enable continuous queries.

As a query language for our framework, an extension of the semantics of CQL is required to include the notion of virtual tables and the associated processing techniques for virtual tuples.

However, the introduction of virtual tables raises the need to define a new functionality: expressing optimization criteria to choose the optimal tuple(s) among a group of possibilities. We need to choose the optimal virtual tuple corresponding to an event so that only the "optimal" service is actually invoked. We present a solution to this need through a new clause in SQL: the COLLAPSE clause.

Example 6 *For the night surveillance scenario, we need to handle events, represented as tuples in the "sensor" table. In order to take a photo of the event location, those tuples have to be associated with a "camera" service, represented as tuples in the "camera" virtual table. More than one service may be able to take the photo. However, only one photo is needed: the system should select the "optimal" service,* i.e. *the service with the least estimated response time. The definition of "optimal" is context-dependent: it justifies the introduction, at the declarative level, of a new clause in SQL.*

4.1. *Continuous query processing with virtual tables*

4.1.1. *Taking into account virtual tables*

All data sources are represented as virtual tables associated with binding patterns. Non-virtual tables are only extreme cases with zero binding pattern. In a logical query plan, intermediary tables between operators are also virtual tables as well as the output table of the root operator.

After a query is parsed, its semantics is checked using the metadata catalog referencing the names and properties for tables and attributes. It is then transformed into a logical query plan of operators like joins, selections, projections, aggregations.

The metadata catalog also contains the binding patterns associated with virtual tables. A specific operator, the dependent join (Florescu *et al.*, 1999), is required to realize a binding pattern: it provides values for the binding pattern input attributes (by an equality predicate with another attribute or a constant value) and allows to retrieve the values for the binding pattern output attributes. Binding patterns add constraints on the join order for the tables: a dependent join operator should have values for its input attributes, so other dependent joins that retrieve those values (as the output attributes of their binding patterns) should occur before.

A dependent join operator produces an output table containing virtual tuples with values for the binding pattern input attributes. However, it is not already necessary to invoke the service function associated with the binding pattern to retrieve the output attribute values. On the contrary, it is interesting to keep the tuples as long as possible in a virtual form (with no values for the output attributes), in order to make asynchronous calls (Goldman *et al.*, 2000) to the functions and speed up the global query processing.

Two additional logical operators need to be integrated in the operator tree for each required binding pattern. An *invocation operator* makes asynchronous calls to the function associated with the binding pattern, and a *binding operator* actually sets the requested values into the tuple attributes. Note that the invocation operator is not blocking for the tuples whereas the binding operator can block a tuple as long as the corresponding asynchronous call has not returned its result lines. The blocking operator ensures that the virtual attributes involved in the binding pattern have their actual values for every output tuple it produces. In (Goldman *et al.*, 2000), the binding operator (called "Request Synchronizer") is present but the invocation operator is integrated in the table scan operator for the data source. The independence of the invocation operator allows a more flexible query plan and leads to further optimization possibilities.

Query optimizations techniques can be applied on the logical query plan. Operators can be reorganized in order to minimize the number and size of tuples to process, e.g. by pushing selection operators down before joins or introducing projections. The number of function calls can also be minimized, e.g. by pushing selection operators down before invocation operators. Further optimization techniques can be applied to the physical representation of the query plan, like merging some operators, in order to compute an optimal physical query plan.

4.1.2. *Continuous query execution*

In the execution phase, the query processor actually executes the physical query plan. Whereas in traditional DBMS, the query processor executes a query plan once to produce a result table, the continuous query processor needs to schedule every operator in (near) real-time, in order to process new tuples from the data streams and insertions/deletions of tuples from the relations, and to propagate them through the operator tree. (Arasu *et al.*, 2003) studies some scheduling algorithms for this context.

In order to realize the binding patterns, the virtual tuple processing technique follows the same principle as the *asynchronous iteration* technique in (Goldman *et al.*, 2000). When processed by a *binding operator*, an input virtual tuple may be duplicated according to the number of result lines for the corresponding function call: each result line will produce one output tuple. Every output tuple contains a copy of all the attribute values from the input virtual tuple, including the input attributes of the binding pattern. It also contains the values for the output attributes of the binding pattern that are retrieved from the result line. The output tuples are virtual in the general case: the output table of the operator may still contain some binding patterns for other virtual attributes.

Example 7 (Using a dynamic virtual table) *In Table 8, a SoCQ allows to handle events from the "sensor" stream (see Table 3): each tuple that has a "accel_x"value greater than 500 is associated with every service from the virtual table "camera" (defined in Example 3 and 4) that covers its location. This coverage is indicated by the boolean virtual attribute "coverage"provided by the service function* checkCover-

age(). *The virtual attribute "photo" represents an actual photo provided by the service function* takePhoto().

Table 8. *Example of a query using the virtual table "camera" and the result table at different timestamps*

```
SELECT sensor.id,sensor.location,camera.id,camera.photo
FROM sensor,camera
WHERE sensor.accel_x > 500.0
  AND sensor.location = camera.location
  AND camera.coverage
```

Timestamp @ 25	Timestamp @ 30
(65, 'e', 2, BLOB("photo001.jpg")) @ 25	(65, 'e', 2, BLOB("photo001.jpg")) @ 25
(65, 'e', 3, BLOB("photo002.jpg")) @ 25	(65, 'e', 3 ,BLOB("photo002.jpg")) @ 25
	(17, 'd', 3, BLOB("photo003.jpg")) @ 27
	(17, 'd', 5, BLOB("photo004.jpg")) @ 27
	(17, 'd', 8, BLOB("photo005.jpg")) @ 27
	(98, 'c', 5, BLOB("photo006.jpg")) @ 28

4.2. *The COLLAPSE clause*

Virtual tables provide a mean to represent services that are dynamically discovered in a pervasive environment. In Example 7, each tuple from the "sensor" stream is joined with every tuple from the "camera" virtual table, *i.e.* all available services. Even if a condition on the coverage allows to discard some tuples, the result table may contain several tuples corresponding to one event: with the binding patterns, the system has to invoke the *takePhoto()* function for several services. Although this behavior may be wanted, the goal of the night surveillance scenario is to choose the best way to handle each event, *i.e.* to call only the best service to handle an event. With the "camera" virtual table, the best service for a given location is the one with the minimum value for the 'cost' virtual attribute.

SoCQs may need to explicitly express criteria to choose the optimal service for each event. From a data-centric point of view, the goal is to extract the first tuple from a group of tuples according to a given ordering. On the one hand, it is similar to the definition of a top-K query (here with K=1) applied to sub-groups of tuples. On the other hand, computing one tuple from a group of tuples is similar to an aggregation.

However, standard aggregation functions like MIN, MAX or AVG, accept only one parameter and return only one value. Some DBMS like PostgreSQL allow to define User Defined Aggregates (UDAs) that accept several parameters, but still return one value. Even if the return value may be composite, *i.e.* a structure composed of several attributes, it does not allow a simple syntax to express the required optimization.

In this setting, we propose a new clause for SQL in order to express such an aggregate in a generic and unambiguous way: the COLLAPSE clause. It allows to define an aggregate function returning several attributes that are retrieved from the optimal tuple for each group. Table 9 shows the syntax of the COLLAPSE clause. It has to immediately follow the GROUP BY clause.

Table 9. *Syntax of the* COLLAPSE *clause*

```
SELECT ...
FROM ...
WHERE ...
GROUP BY groupAtt1, groupAtt2, ...
COLLAPSE (att1,att2,...,attN) INTO name
  USING orderAtt1 [ASC|DESC], orderAtt2 [ASC|DESC], ...
HAVING ...
```

The set of attributes ("att1", "att2", ..., "attN") are the *collapsed attributes* returned by the aggregate function. The optimal tuple corresponds to the first tuple of the group when it is ordered according to the USING part (like with an ORDER BY clause in SQL). The INTO part defines the name for the set of collapsed attributes, so that they can be referenced as "name.attribute" in the SELECT clause and/or the HAVING clause. Collapsed attributes can thus be used like other standard aggregate values in these both clauses.

Example 8 (Using a COLLAPSE clause) *In Table 10, a* COLLAPSE *clause extracts for each group ("s.id", "s.location") the tuple that minimizes the "c.cost" value,* i.e. *the first tuple in each group ordered by the "c.cost" value in ascending order. The name of this collapsed set is "bestCamera": the collapsed attributes are identified by "bestCamera.cost" and "bestCamera.photo" in the* SELECT *clause and in the* HAVING *clause.*

Table 10. *Example of a query using a* COLLAPSE *clause*

```
SELECT s.id,s.location,bestCamera.cost,bestCamera.photo
FROM sensor s, camera c
WHERE s.location = c.location
  AND c.coverage
GROUP BY s.id, s.location
COLLAPSE (c.cost, c.photo) INTO bestCamera
  USING c.cost ASC
HAVING bestCamera.cost < 5
```

Although we present this clause in the context of SoCQs to choose the optimal service(s) to be called for a given event, it can be applied to other cases, in particular in non-continuous queries, e.g. in multi-objective queries (Balke *et al.*, 2004) or to declaratively define complex aggregations like in (Akinde *et al.*, 2001; Chatziantoniou, 1999).

5. Implementation

Continuous query processing techniques are inspired from standard query processing techniques (Garcia-Molina *et al.*, 1999). However, the introduction of the notion

of time impacts on the whole conception. We propose an architecture of a SoCQ-enabled DSMS, inspired by an open-source DSMS: STREAM (Arasu *et al.*, 2003), whose prototype has been developed at Stanford University. We first briefly describe the STREAM prototype. We explain the different entities used by a SoCQ processor and describe the architecture of our SoCQ processor prototype. The implementation of the new query operators is tackled in details. We then describe first experimental results from our prototype.

5.1. *The STREAM prototype*

STREAM provides support for "a large class of declarative continuous queries over continuous streams and traditional stored data sets" (Arasu *et al.*, 2003). It is composed of a CQL parser, a query analyzer that produces execution plans, and a plan manager that schedules operators to execute the continuous queries. Execution plans are optimized at the logical level, then at the physical level. The prototype allows to register relations and streams schemas, and to associate them with a physical data source. A physical data source is an interface (in C++) that is currently implemented as a file reader for both relations and streams. Support for four data types is provided: byte, integer, float, and fixed-length string.

In the current implementation, CQL allows to define queries similar to SQL: SELECT – FROM – WHERE – GROUP BY. The FROM clause is extended to define windows over the streams. The relation-to-stream operators (IStream, DStream, RStream) are expressed by a keyword with parenthesis surrounding the whole query text. Aggregation functions are limited to the MIN, MAX and AVG functions over integer and float attributes.

5.2. *SoCQ processor entities*

The goal of the SoCQ processor is to execute continuous queries over data relations and data streams, with service calls as additional data sources. Like in STREAM (Arasu *et al.*, 2003), relations and streams are represented with a unifying table entity, and table data is considered as a flow of tuple insertions and corresponding tuple deletions, called elements. Query operators work with element queues as input, and produce elements into an output queue.

However, whereas the STREAM prototype considers relations and streams only as element queues (all elements are discarded when they are consummed), the SoCQ processor differentiates between the two types of tables: relations keep their current content (inserted tuples not yet deleted) and can provide them to later queries.

The SoCQ processor also manages binding patterns for virtual tables. A virtual table can have several binding patterns. A binding pattern associates the service reference attribute of the virtual table and a service interface function, and maps some attributes of the virtual table to the input and output parameters of the function.

Services are external entities implementing some functions (currently, a set of shell scripts) and are mapped to some service interfaces, indicating that the service implements all the functions of those interfaces. A service reference attribute of a virtual table contains an identifier of a service: a binding pattern can be realized for every tuple from this table by calling the associated function of the referenced service.

As virtual attributes of tuples from a virtual table do not have values until they reach a binding operator in a query, they don't need to be physically represented in the source table. A virtual table then has two relation schemas: its main schema, associated with binding patterns, and an internal schema, containing only the non-virtual attributes. The physical representation of the data of a virtual table is based on this internal schema. The main schema and its associated binding patterns are used at a logical level to compute query plans.

A query plan represents a SoCQ and is composed of several query operators linked by element queues to other operators or directly to tables. Query operators are:

– relational operators: selections, projections, joins and aggregations (also managing the COLLAPSE clause);

– operators specific to streams: istreams and windows;

– operators specific to binding patterns: invocations and bindings.

5.3. *SoCQ processor architecture*

The architecture of the SoCQ processor is designed to handle the different entities needed to process SoCQs: tables with binding patterns, service interfaces, services, and query plans. It is composed of seven main modules, as shown in Figure 1:

– the *System Interface*: this module is an interface for system administration. It parses the user commands and interacts with the *System Catalog* (table management commands, service interface management commands) or with the *Service Manager* (service registration commands, service mapping commands). It also handles the user queries: the SoCQs are parsed and then routed to the *Query Plan Manager*;

– the *System Catalog*: this module allows to register the tables and the service interfaces. A table is associated with a name, a data schema and binding patterns. Its internal data schema is also computed. A service interface is identified by a name and contains a set of functions with their description: function name, input and output parameters, output cardinality. When a table is registered, the *Table Manager* is notified to physically create the table;

– the *Service Manager*: this module allows to register the services and their mappings to service interfaces. A service is associated with a physical access method (e.g. executing a shell script) and a physical service identifier (e.g. a shell script name). The module can asynchronously call a service function through a mapped interface function with some given values for the input parameters and return the output parameters values;

– the *Table Manager*: this module manages the physical tables created through the *System Catalog*. It allows to connect element queues to a table output, and to access to a table input element queue in order to insert and delete tuples. It computes the state of the tables from their input element flow (tuple insertion, tuple deletion) and forwards those elements to the connected element queues;

– the *Storage Manager*: this module is responsible for the storage of the table content: the tuples. It allocates some memory space for each table and manages the insertion of new tuples. When a tuple is deleted from its table, it may be not immediately deleted from memory: the module maintains a reference count for each tuple so that its memory space can be released only when it is no longer needed;

– the *Data Socket Manager*: this module manages external connections to table output and input through network sockets. It uses a simple dedicated protocol to send and receive element flows. It interacts with the *Table Manager* to connect to the tables;

– the *Query Plan Manager*: this module interprets SoCQs and optimizes the queries into physical query plans. A physical query plan is a tree of query operators whose leaf operators are connected to the output of the involved tables and the tree root operator feeds the input of the result table. Some intermediary tables can be created for operators that need to create tuples (projection, join…). The module continuously executes the registered query plans, *i.e.* scheduling every query operator in (near) real-time, and can dynamically register new query plans or unregister some existing ones.

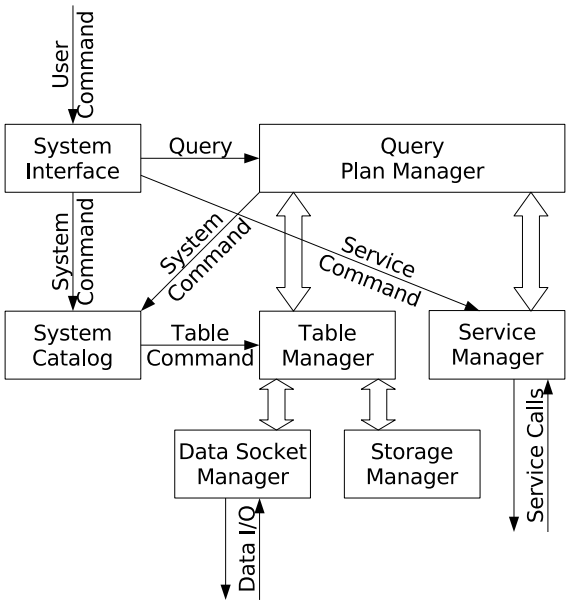

Figure 1. *Architecture of the SoCQ processor*

5.4. *Focus on new operators*

In order to handle SoCQs, we need to implement the COLLAPSE clause into the aggregation operator, and to develop two new operators dedicated to the realization of binding patterns: the invocation operator and the binding operator.

The COLLAPSE clause is an extended aggregation function: it computes some aggregated values for a group of tuples. The aggregation operator seems then fitted for the task. However, as those values are taken from the optimal tuple based on some criterion, the operator needs to find this optimal tuple every time the group of tuples is modified (insertion, deletion).

We have implemented this functionality by maintaining a list of all tuples sorted by the optimality criterion, so that the first tuple found for each group is the optimal tuple for this group. A newly inserted tuple must be sorted in the list, but the order is not modified by a deletion. Standard aggregation functions, like SUM, MIN, MAX, can be computed on the sorted list of tuples as in a standard aggregation operator. Collapsed values are copied from the optimal tuple of a group.

In order to realize the binding patterns, service functions should be called and tuples should be filled in with the result data. Furthermore, asynchronous calls allow the system to process tuples from other operators or to make other asynchronous calls while current calls are pending.

We have implemented this functionality with two operators. The invocation operator is configured to call a service interface function. It needs to extract the service reference attribute and the attributes forming the input parameters from each input tuple. It can then launch the corresponding asynchronous calls through the *Service Manager*. Each call is identified with the tuple identifier so that the binding operator can match tuples with their corresponding call result. The invocation operator, after launching a call, forwards the tuple via its output element queue to the next operator.

The binding operator receives the input tuples and blocks them until their corresponding result set is provided by the *Service Manager*. It can then produce the resulting tuples. However, as the calls are asynchronous, the call results may arrive in a random order: the operator needs to ensure that the produced tuples still follow the timestamp order.

5.5. *Experimentation*

The whole architecture has been implemented in C++ on a LINUX machine. We choose to experiment the night surveillance scenario from the motivating example with a query similar to Example 8. The actual query is shown in Table 11: it involves the stream “sensor”, the virtual table “camera”, and a COLLAPSE clause. The window specification “[now]” indicates that a tuple from the “sensor” table will not be joined

with tuples inserted at a later timestamp in the other table. The IStream operator indicates that the output of the query is a stream: output tuples will never be deleted.

Table 11. *Service-oriented Continuous Query for the experimentation of the night surveillance scenario*

```
SELECT ISTREAM s.location, best.id, best.cost, best.photo
FROM sensor s [now], camera c
WHERE s.location = c.location
GROUP BY s.location
COLLAPSE (c.cost,c.id,c.photo) INTO best
  USING c.cost ASC
```

To evaluate this query, synthetic data have been generated to simulate the environment. For the table “sensor”, 100 random tuples have been generated, with a timestamp between 2 and 99 indicating a “accel_x” value between 100 and 900 and a location label between 10 possibilities (“a” to “j”). The cameras have been simulated by two shell scripts for the two involved functions of the camera interface: *getCost()* and *takePhoto()*. The two scripts takes the “location” attribute as an input parameter. The *getCost()* script returns a random cost value, and the *takePhoto()* script returns the location label in uppercase, in order to prove an actual data processing made by the function calls.

The query from Table 11 corresponds to the logical query plan in Figure 2. The table “camera” and the windowed table “sensor” are joined by a Cartesian product. Note that the predicate “s.location = c.location” is not a join predicate, but an indication for the realization of the binding patterns: the input virtual attribute “location” from the table “camera” is then an alias for the non-virtual attribute “location” from the table “sensor”.

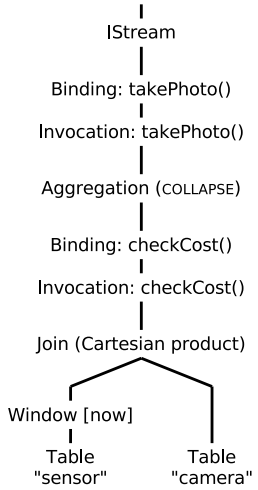

Figure 2. *Logical query plan for the night surveillance scenario*

The following operators in the logical query plan are the invocation and binding operators that realize the first binding pattern: it provides the “cost” attribute, by calling the *getCost()* function using the service reference attribute “id” from the table “camera”. The aggregation operator can then group the tuples by “location” and extract the optimal tuple according to the minimum “cost” attribute. Another pair of invocation and binding operators realizes the second binding pattern: it provides the “photo” attribute for the optimal tuples generated by the aggregation operator, by calling the *takePhoto()* function.

In order to execute this query in the SoCQ Processor, several steps need to be done to prepare the system:

1) launching the SoCQ processor,

2) registering a service interface “iCamera” with the two functions *getCost()* and *takePhoto()*,

3) creating the two tables “sensor” and “camera” with associated binding patterns,

4) registering some services (executing the shell scripts), and mapping them to the interface “iCamera”,

5) registering the query.

The SoCQ processor is now executing the query. We manually insert one tuple for each registered service into the table “camera” (two services in the actual experimentation). Using a rudimentary interface tool (Figure 3), we connect the table viewer (on the left side in the interface) to the query output table and we insert the randomly-generated tuples (on the right side in the interface) into the table “sensor”. The interface tool enables to insert elements from an input file at three different speeds: element by element, all the elements at the current timestamp, all the elements (until the end of the input file). It also enables to save the query output table into an output file.

All the tuples from the input file have been inserted, timestamp after timestamp. Tuples have been progressively retrieved from the query output table into the table viewer and written into the output file. The beginning of both files are presented in Table 12, showing the elements inserted into the table “sensor” and the elements retrieved from the query output table between timestamps 1 and 10.

The query output table content is as expected: for each timestamp, the tuples are grouped by the ‘location’ attribute and the value of the ‘photo’ attribute corresponds to the processing of the value of the ‘location’ attribute by the *takePhoto()* function.

Additional experiments have been scheduled to test the prototype with more complex queries and larger data sets.

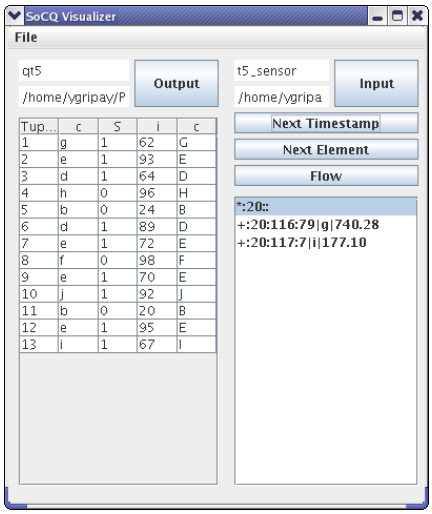

Figure 3. *Snapshot of the interface tool. The table on the left side represents the output table of the query. The list of elements on the right side contains the next elements from the input file to be inserted into the "sensor" table*

Table 12. *Beginning of the files containing the tuples to be inserted into the "sensor" table (left column) and the tuples retrieved from the query output table. The first line is the schema of the tuples: (integer, char, real) for the table "sensor" and (char, service reference, integer, char) for the query output table. The following lines are elements represented as <type>:<timestamp>:<tuple ID>:<tuple>: elements with '+' are insertion elements, elements with '*' are heartbeat elements, indicating a change of timestamp*

```
icr
*:2::
*:3::
+:3:101:52|g|772.66
*:4::
+:4:102:70|e|789.46
+:4:103:37|e|426.09
*:5::
+:5:104:40|h|574.23
+:5:105:2|h|193.28
+:5:106:44|d|871.31
*:6::
+:6:107:89|b|441.39
*:7::
*:8::
*:9::
+:9:108:84|e|841.65
+:9:109:75|d|214.01
*:10::
```

```
cSic
*:3:0:
+:3:1:g|1|62|G
*:4:0:
+:4:2:e|1|93|E
*:5:0:
+:5:3:d|1|64|D
+:5:4:h|0|96|H
*:6:0:
+:6:5:b|0|24|B
*:7:0:
*:8:0:
*:9:0:
+:9:6:d|1|89|D
+:9:7:e|1|72|E
*:10:0:
```

6. Conclusion

In this paper, we have presented our ongoing work on the framework for *Service-oriented Continuous Queries* (SoCQs) that enables to build queries over relations, streams and services. It is built on top of the CQL specifications (Arasu *et al.*, 2003) that manage streams and relations.

The SoCQ framework introduces tables and virtual tables as a unified mean to represent relations, streams and services. A virtual table has virtual attributes and is related to a service interface, using binding patterns to indicate which virtual attributes should be used as an input for a service function call or retrieved as an output from a service function call result. At the logical query plan level, a dependent join operator provides values for the input virtual attributes from other non-virtual attributes. During query execution, an invocation operator makes asynchronous calls to functions in a non-blocking manner, and a binding operator is used to block until the data are effectively retrieved from the function calls. The underlying principle of virtual tables can be used as a mean to take in charge the dynamicity of pervasive environments where services appear and disappear.

Many services may be able to provide a virtual attribute value for a specific query. We have thus introduced the COLLAPSE clause that declaratively defines a criterion for the selection of a sub-set of service function calls. The COLLAPSE clause extracts the top-K tuples from a group of tuples according to a given ordering. It intends to replace and augment the procedural and ad hoc user-defined aggregates that are available today.

We have also presented our SoCQ processor prototype, inspired by the STREAM prototype (Arasu *et al.*, 2003). Our prototype handles SoCQs over virtual tables representing relations and streams. It also manages both the COLLAPSE clause and the binding pattern mechanism.

The experimentation have presented the execution of a SoCQ from the running example of our article using synthetic data and services simulating devices. It has demonstrated the capabilities of the SoCQ processor and the power of expression of SoCQs. In future work, we plan to continue the development of the prototype in order to optimize the implementation of the query operators and to improve the (currently basic) query optimizer. Furthermore, we aim to develop a benchmark involving real data sets and services from a pervasive environment.

7. References

Abadi D. J., Ahmad Y., Balazinska M., Cetintemel U., Cherniack M., Hwang J.-H., Lindner W., Maskey A. S., Rasin A., Ryvkina E., Tatbul N., Xing Y., Zdonik S., " The Design of the Borealis Stream Processing Engine", *CIDR 2005, Proceedings of Second Biennial Conference on Innovative Data Systems Research*, 2005.

Akinde M. O., Chatziantoniou D., Johnson T., Kim S., " The MD-Join: An Operator for Complex OLAP", *ICDE'01: Proceedings of the 17th International Conference on Data Engineering*, p. 524, 2001.

Arasu A., Babcock B., Babu S., Datar M., Ito K., Motwani R., Nishizawa I., Srivastava U., Thomas D., Varma R., Widom J., " STREAM: The Stanford Stream Data Manager", *IEEE Data Engineering Bulletin*, vol. 26, n° 1, p. 19-26, 2003.

Balke W.-T., Güntzer U., " Multi-objective Query Processing for Database Systems", *VLDB'2004: Proceedings of the 30th International Conference on Very Large Data Bases*, p. 936-947, 2004.

Barga R. S., Chkodrov G., " Coping with Variable Latency and Disorder in Distributed Event Streams", *ICDCSW'06, Proceedings of the 26th IEEE International Conference on Distributed Computing Systems Workshops*, 2006.

Becker C., Handte M., Schiele G., Rothermel K., " PCOM – A Component System for Pervasive Computing", *PerCom'04, Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, p. 67, 2004.

Brumitt B., Meyers B., Krumm J., Kern A., Shafer S., " EasyLiving: Technologies for intelligent environments", *HUC 2000, Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, p. 12-29, 2000.

Chandrasekaran S., Cooper O., Deshpande A., Franklin M. J., Hellerstein J. M., Hong W., Krishnamurthy S., Madden S., Raman V., Reiss F., Shah M., " TelegraphCQ: Continuous Dataflow Processing for an Uncertain World", *CIDR 2003, Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

Chatziantoniou D., " The PanQ tool and EMF SQL for Complex Data Management", *KDD'99: Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, p. 420-424, 1999.

Chen J., DeWitt D. J., Tian F., Wang Y., " NiagaraCQ: A Scalable Continuous Query System for Internet Databases", *Proceedings of ACM SIGMOD International Conference on Management of Data*, p. 379-390, 2000.

Cherniack M., Balakrishnan H., Balazinska M., Carney D., Çetintemel U., Xing Y., Zdonik S., " Scalable Distributed Stream Processing", *CIDR 2003, Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

Ding L., Rundensteiner E. A., " Evaluating Window Joins over Punctuated Streams", *CIKM'04, Proceedings of the 13th ACM international conference on Information and Knowledge Management*, p. 98-107, 2004.

Estrin D., Culler D., Pister K., Sukhatme G., " Connecting the Physical World with Pervasive Networks", *IEEE Pervasive Computing*, vol. 1, n° 1, p. 59-69, 2002.

Florescu D., Levy A., Manolescu I., Suciu D., " Query Optimization in the Presence of Limited Access Patterns", *SIGMOD'99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, p. 311-322, 1999.

Franklin M., Halevy A., Maier D., " From Databases to Dataspaces: a new Abstraction for Information Management", *SIGMOD Rec.*, vol. 34, n° 4, p. 27-33, 2005a.

Franklin M. J., Jeffery S. R., Krishnamurthy S., Reiss F., Rizvi S., Wu E., Cooper O., Edakkunni A., Hong W., " Design Considerations for High Fan-In Systems: The HiFi Approach", *CIDR 2005, Proceedings of Second Biennial Conference on Innovative Data Systems Research*, 2005b.

Garcia-Molina H., Widom J., Ullman J. D., *Database System Implementation*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

Garlan D., Siewiorek D. P., Smailagic A., Steenkiste P., " Project Aura: Toward Distraction-Free Pervasive Computing", *IEEE Pervasive Computing*, vol. 1, n° 2, p. 22-31, 2002.

Goldman R., Widom J., " WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web", *Proceedings of ACM SIGMOD International Conference on Management of Data*, p. 285-296, 2000.

Grimm R., Davis J., Lemar E., Macbeth A., Swanson S., Anderson T., Bershad B., Borriello G., Gribble S., Wetherall D., " System Support for Pervasive Applications", *ACM Transactions on Computer Systems*, vol. 22, n° 4, p. 421-486, November, 2004.

Pigeot C.-E., Gripay Y., Scuturici M., Pierson J.-M., " Context-Sensitive Security Framework for Pervasive Environments", *ECUMN'07: Fourth European Conference on Universal Multiservice Networks*, p. 391-400, 2007.

Srivastava U., Munagala K., Widom J., Motwani R., " Query Optimization over Web Services", *VLDB 2006, Proceedings of the 32nd International Conference on Very Large Data Bases*, p. 355-366, 2006.

Tian F., DeWitt D. J., " Tuple Routing Strategies for Distributed Eddies", *VLDB 2003, Proceedings of the 29th International Conference on Very Large Data Bases*, p. 333-344, 2003.

Weiser M., " The Computer for the 21st Century", *Scientific American*, vol. 265, n° 3, p. 94-104, September, 1991.

Xue W., Luo Q., " Action-Oriented Query Processing for Pervasive Computing", *CIDR 2005, Proceedings of the Second Biennial Conference on Innovative Data Systems Research*, 2005.

Yao Y., Gehrke J., " Query Processing in Sensor Networks", *CIDR 2003, Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.