Towards Action-Oriented Continuous Queries in Pervasive Systems

Yann Gripay, Frederique Laforest, Jean-Marc Petit
Université de Lyon, INSA-Lyon, LIRIS – UMR 5205 CNRS
7 avenue Jean Capelle
69621 Villeurbanne cedex, France
Yann.Gripay,Frederique.Laforest,Jean-Marc.Petit@insa-lyon.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 to deal with new properties of such information systems including dynamicity, autonomy and decentralization.

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. Those continuous queries could express an event management functionality (e.g. event filtering, event composition), associate events with data from legacy systems, and perform cost-based optimal calls to services.

In this paper, we define virtual tables with binding patterns to represent services of the pervasive environment. By the way, relations, data streams and virtual tables 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.

A prototype on top of STREAM, a DBMS devoted to data streams, has been devised on which first experiments have been carried out on synthetic data.

Keywords Continuous Query; Query Optimization; Non-Conventional Data Sources; Data Streams; Services; Pervasive Systems

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 world-wide 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.

Data Streams 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* [10], 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 [1, 3, 8, 11, 15, 23, 26].

Services With the development of autonomous devices and location-dependent functionalities, information systems tend to become what Mark Weiser [24] called *ubiquitous systems*, or *pervasive systems*. Pervasive systems [6, 7, 17, 19, 20] 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. Service discovery is a common issue [27] in distributed systems (pervasive systems, grids, or even Internet), that tackles service representation, knowledge sharing, and remote execution. In dynamic environments like pervasive systems, the discovery should also be dynamic in order to reflect the currently available services.

As devices may be sensors or actuators [13], services may represent some interactions with the physical environment, like taking a photo from a camera or displaying a picture on a screen. These *actions* bridge the gap between the computing environment and the user environment, and can be managed by the pervasive system. This paper will not consider service discovery techniques, but will consider a way to use and compose services with the notion of *action-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 query processing techniques need to be adapted to those data sources. 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 [25]. 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, or even a group optimization to benefit from parallelism.

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 actions and the predicates. 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 [25], 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: the action to take a photo, the action to send a photo, and a predicate that checks that a camera is able to take a photo of a location. We made a synthesis of this environment in Figure 1.

The continuous query for the night surveillance scenario is given in AortaSQL [25] in Figure 2: an *Action Query* called “night_surveillance” is active from midnight to 6:00 am every day (cf. START and STOP clauses).

```
Data Sources:
  relation phone( id, owner, number );
  stream sensor( id, accel_x, location );
  virtual device table camera( id, ip_address );

Functions:
  photo(camera_ip, location, directory_name) :
    file_name
  sendphoto(phone_number, file_name) : void
  coverage(camera_id, location) : boolean
```

Figure 1: The environment for the night surveillance scenario from Aorta [25]

```
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
```

Figure 2: The query for the night surveillance scenario from Aorta [25]

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 action (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 [3, 26, 15, 8].

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 [12] or relation-to-stream operators [3], 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 action like in Aorta.

1.2 Evolution of Continuous Queries

In this paper, we present an ongoing effort to develop a framework for *Action-Oriented Continuous Queries* (AOCQs), whose aim is to integrate services, *i.e.* distributed functionalities, in continuous queries over data streams. AOCQs 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 [3]. We represent sets of similar services as virtual tables containing a tuple per service and associated with one or more binding patterns [14, 18, 22] 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.

AOCQs can imply services that are statically bound [18, 22] or dynamically discovered in the pervasive information system, like in [21]. The optimal services (at a given time for a given data set) are selected and called during query execution. AOCQs can then express an event management functionality like event filtering and composition, and perform cost-based optimized calls to services. In pervasive environments, those queries can use services of the pervasive system to execute actions: continuous queries can smoothly evolve from data-oriented queries to action-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 SQL clause, called the COLLAPSE clause, is proposed to define an optimizing criterion over groups of tuples.

- The development of a prototype of a query processor for AOCQs, from which first experimental results over synthetic data are described. For the time being, the AOCQ processor is built on top of the STREAM prototype [3], a DSMS developed at Stanford University, and allows to show both the power of expression of AOCQs 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

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 [3]. This widely adopted model [23, 26, 11, 1, 8, 15] 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 [5].

Various data sources may generate data streams: *e.g.* sensors, that ranges from physical sensors (light, temperature...) to logical sensors (network monitoring, applications...). However, some other data may be considered as streams: transfers of large tuple sets from distributed databases are equivalent to data streams [22], even if they are bounded streams.

Costly Data Sources Data streams are often seen as virtual relational tables, but the generation of the tuples depends on several factors that can make it slow or unsafe. Some tuple attributes may also be expensive (in term of time or resources) to be acquired.

Unsafe data sources like sensor networks may introduce some latency and disorder for the tuples. Synchronization among events from different streams may then be erroneous. [5] proposes a system to cope with these problems thanks to a stream conditioning mechanism that reorders and synchronizes tuples.

Some safe data sources or function evaluations may also be slow, like web services or sensed attributes. Introducing asynchronous calls to data sources and synchronization operators in query execution plans, like in [18], allows to process incomplete tuples until their costly attributes are required, which gives time to complete the asynchronous call and fill in the missing attribute values. [25] 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 executing an action in a pervasive environment. Furthermore, group optimization allows to optimally distribute simultaneous function evaluations among the possible candidates.

Continuous Query Definition 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. [3] 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 [26, 1, 15] do not express the difference between operator categories, and work, in their semantics, only with data streams.

Selection and projection operators keep a clear semantics with data streams. Selection operators filter tuples based on the values of their attributes, and projection operators keep only a subset of the attributes of tuples. There is no difference between data streams and relational tables for these operators.

With continuous queries, data streams are supposed to be unbounded. However, aggregation operators for relations need to have a complete view of all tuples (*e.g.* the COUNT operator in SQL), which is impossible for unbounded streams. A mechanism of punctuations [12], indicating the end of a group of related tuples, is needed in order to allow the aggregation operator to output its resulting aggregated tuples, thus creating an aggregated stream. In [3], aggregation operators are seen as relation-to-relation operators: the transformation of the input stream into a relation is done by other operators, and the output is a time-varying relation.

Join operators face the same problems as aggregation operators. 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 coming from the other stream. The sets of tuples should then be bounded. A window defines a bounded subset of tuples from a stream (it is the only stream-to-relation operator in [3]), based on time or on the number of tuples. Sliding windows [3, 12] have a fixed size and continuously move forward (*e.g.* the last 100 tuples, tuples within the last 5 minutes). Hopping windows [26] have a fixed size and move by hop, defining a range of interval (*e.g.* 5-minute window every 5 minutes). In [8], 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. [3] defines also 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 [3] 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 [3, 26, 15, 8] propose an extension of SQL in order to work with both relational databases and data streams. Some articles [10] tackle continuous querying over distributed XML data sets and propose an extension of XML-QL. Others [1] 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 represented using a relation schema, like for relational tables.

The traditional query structure (SELECT – FROM – WHERE) can still express selection, projection, join, even aggregation (GROUP BY – HAVING), except that the FROM clause contains references to streams. The main extension is the definition of windows for the streams. In some articles [3, 15], window specifications are added in the FROM clause for each stream, defining the time-based or count-based size. Other extensions [26, 8] add a clause to express a global window for every stream of a query (only time-based).

As continuous queries may be running forever, a continuous query management system should allow an administrator to express when and how long a query should be activated. Some extensions propose commands (like SQL commands CREATE, DROP) to activate and deactivate queries [10, 15], whereas others [8, 25] integrate clauses in the language to express start time and expiration time.

Continuous Query Processing The long-running nature of continuous queries changes the definition of execution plans. An execution plan is composed of operators that may handle data streams, making the execution more dynamic than for standard queries. Two methods appear in the literature to cope with this dynamicity.

The first method is the construction of a global execution plan, like in [3, 15, 26, 1], 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 (on the same streams) among the different queries.

The second method is to dynamically distribute tuples to one of their possible next operators, each tuple creating its own execution plan. Operators (called Eddies [8]) are responsible of the choice of the destination for each tuple they have processed, depending on the dynamic state of other candidate operators.

The optimization process always depends on a chosen cost metric. In traditional DBMS, standard queries are often optimized based on the total execution time of the query. This cost metric is no longer available for continuous queries due to their long-running nature. Other cost metrics are then proposed in the literature. The processing time by tuple seems to be the natural extension of the previous cost metric adapted to data streams. Another approach is the bottleneck metric [22] that optimizes the throughput of the queries.

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 with some renewed definitions for relations, streams, 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 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 [3] 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 considered simultaneous if they are both associated with the same timestamp.

In order to homogeneously represent a relation and a stream, 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. It can then homogeneously represent a relation or a stream.

We consider the *instantaneous relation* [3] 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*
Figure 3 and Figure 4 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 phone( id INTEGER, owner CHAR(10),
                 number CHAR(10))

Timestamp @ 25
(34,"Alice","069911XXXX") @ 10
(25,"Bob"  ,"069922XXXX") @ 12

Timestamp @ 30
(34,"Alice"  ,"069911XXXX") @ 10
(18,"Charlie","069933XXXX") @ 26
(24,"David"  ,"069944XXXX") @ 28
```

Figure 3: Schema and two instantaneous relations at different timestamps for the table representing the "phone" relation

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

Timestamp @ 25
(18, 362.15, 'a') @ 16
(65, 569.42, 'e') @ 25

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

Figure 4: Schema and two instantaneous relations at different timestamps for the table representing the "sensor" stream

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*
Figure 5 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.*

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

Figure 5: Example of Service Interface

To smoothly integrate services in our framework, we propose to use the notion of *binding patterns*. 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" [14]. A relation with binding patterns can represent an external data source with limited access patterns [14] in the context of data integration. It can also represent an interface to an infinite data source like a web site search engine [18], 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 [22].

In our framework, we propose to define a *virtual table* using a service interface as a generalization of a table: its schema can contain *virtual attributes* and is associated with *binding patterns* involving functions from the 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*
Figure 6 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*.*

```
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 )
}
```

Figure 6: Schema and binding patterns for the virtual table "camera"

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. 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, Figure 7 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 no longer belongs to the table at timestamp 30, because the service itself is no longer available in the pervasive environment.*

```
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
```

Figure 7: Two instantaneous relations at different timestamps for the virtual table "camera"

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 as 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 [14, 18, 22]. 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 Scenario*
The environment for the night surveillance scenario is represented in Aorta [25] with a relation, a stream, a virtual device table and three functions (Figure 1). Using our framework, it 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", defined in Figure 8, representing a function that sends a MMS (Multimedia Message) to a cell phone. It is statically bound to a service from the environment implementing this function (not represented in the figure).

To end up, virtual tables generalize the notion of tables representing relations and streams. 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 1 summarizes the constraints for each type of data sources.

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 involv-

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

Table 1: Summary of constraints for each type of data sources

```
SERVICE INTERFACE sendMmsInterface {
  FUNCTION send( message CHAR(255),
                 picture BLOB,
                 destination CHAR(10)) :
             ( status BOOLEAN )
}

VIRTUAL TABLE sendMMS( text CHAR(255) VIRTUAL,
                       image BLOB VIRTUAL,
                       phone_no CHAR(10) VIRTUAL,
                       result BOOLEAN VIRTUAL )

BINDING PATTERNS FOR sendMMS
    USING sendMmsInterface {
  FUNCTION send( text, image, phone_no ) :
             ( result )
}
```

Figure 8: Schema and binding patterns for the static virtual table "sendMMS"

ing different types of data sources using a single SQL-like declarative language, without worrying about the particular implementations of the data sources.

4 Query Processing for AOCQs

AOCQs are continuous queries over relations and data streams, with the addition of virtual tables for functions and services. Simple queries could be expressed using a SQL-like declarative language. CQL (Continuous Query Language [3]) provides syntax extensions to SQL in order to handle the specificities of data streams and to allow 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 patterns. 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 [14], 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 [18] 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 tuple whereas the binding operator can block a tuple as long as the 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 [18], 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. For this step, we rely on well-known logical optimization techniques and do not propose new ones.

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 resulting table, the continuous query processor needs to schedule each operators 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. [3] 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 [18]. 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 and Example 8 demonstrate two AOCQs, one involving a static virtual table and one involving a dynamic virtual table.

EXAMPLE 7 *Using a Static Virtual Table*
In Figure 9, an AOCQ expressed in CQL [3] allows to define the following behavior: for each phone, send a MMS containing a "Hello (name) !" message and a "welcome.jpg" image (interpreted as a BLOB constant in the query). The query uses the "phone" table defined in Example 1 and the static virtual table "sendMMS" defined in Figure 8. As the query is continuous, all current and future phones inserted in the "phone" relation will receive a MMS. Note that the tuple corresponding to "Bob" does not belong to the resulting table at timestamp 30 because it is deleted from the "phone" table (see Figure 3). However, the corresponding call to the service function happens at timestamp 12 (when the tuple is inserted in the "phone" table). It is possible to keep a trace of that tuple by requesting the resulting stream of inserted tuples instead of the resulting relation (like with the ISTREAM *keyword in CQL [3]).*

```
SELECT phone.owner, phone.number, sendMMS.result
FROM phone, sendMMS
WHERE phone.number = sendMMS.phone_no
  AND sendMMS.image = BLOB("welcome.jpg")
  AND sendMMS.text = "Hello "||phone.owner||" !"

Resulting Table:

Timestamp @ 25
("Alice"  ,"069911XXXX",true) @ 10
("Bob"    ,"069922XXXX",true) @ 12

Timestamp @ 30
("Alice"  ,"069911XXXX",true) @ 10
("Charlie","069933XXXX",true) @ 26
("David"  ,"069944XXXX",true) @ 28

List of Function Calls:

sendMMS ("Hello Alice !", BLOB("welcome.jpg"),
           "069911XXXX") : (true) @ 10
sendMMS ("Hello Bob !", BLOB("welcome.jpg"),
           "069922XXXX") : (true) @ 12
sendMMS ("Hello Charlie !", BLOB("welcome.jpg"),
           "069933XXXX") : (true) @ 26
sendMMS ("Hello David !", BLOB("welcome.jpg"),
           "069944XXXX") : (true) @ 28
```

Figure 9: Example of a query using the static virtual table "sendMMS"

Example 8 *Using a Dynamic Virtual Table*
In Figure 10, an AOCQ allows to handle events from the "sensor" stream (see Figure 4): 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'. The virtual attribute 'photo' represents an actual photo provided by the service. As the result table is a join between a stream and a virtual table, no result tuple can be deleted: the result table is itself a stream.

```
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

Result (stream):

Timestamp @ 25
(65, 'e', 2, BLOB("photo001.jpg")) @ 25
(65, 'e', 3, BLOB("photo002.jpg")) @ 25

Timestamp @ 30
(65, 'e', 2, BLOB("photo001.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
```

Figure 10: Example of a query using the virtual table "camera"

4.2 The COLLAPSE Clause

Virtual tables provide a mean to represent services that are dynamically discovered in a pervasive environment. In Example 8, 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.

It could be expressed using a nested query as in Figure 11. However, nested queries are not satisfying for this goal as it complexifies the query design. The optimizing criterion is not well identified and may still select several tuples in case of equality.

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

Figure 11: Query that selects the best service using a nested query

AOCQs 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. Furthermore, it requires the development of a new UDA adapted to the type and number of involved attributes for each query. Three functions are required for a UDA: an initialization function that initializes the aggregate state with the first tuple of the group, an iteration function that updates the aggregate state for each following tuple, and a finalization function that returns the aggregated value computed from the aggregate state. UDA function are developed using DBMS-specific language, with a non-declarative approach: query optimization opportunities are thus reduced for the query processor.

EXAMPLE 9 *Using a User-Defined Aggregate over several Attributes*
Figure 12 shows a possible query syntax using such a UDA: the UDA_MIN aggregation function works only for three attributes and returns a composite value containing this three attributes, retrieved from the tuple that minimizes the first attribute. This syntax is ambiguous as it does not show the composite nature of the function output.

```
SELECT s.id, s.location,
       UDA_MIN(c.cost, c.id, c.photo)
FROM sensor s, camera c
WHERE s.location = c.location
  AND c.coverage
GROUP BY s.id, s.location
```

Figure 12: Example of a query using a User-Defined Aggregate UDA_MIN over three attributes

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. Figure 13 shows the syntax of the COLLAPSE clause. It has to immediately follow the GROUP BY clause.

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

Figure 13: Syntax of the COLLAPSE Clause

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 10 *Using a* COLLAPSE *clause*
In Figure 14, 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.*

A collapsed attribute set can be defined as an

```
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
```

Figure 14: Example of a query using a COLLAPSE clause

implicit table whose schema contains the grouping attributes and the collapsed attributes. The *implicit table* contains the collapsed tuples for all groups. The query result is then a join between this *implicit table* and the other tables based on the equality between the grouping attributes. This definition allows a generalization of the COLLAPSE clause: the *implicit table* can contain more than one optimal tuple for a group, as in top-K queries. A query can specify the maximum number K of collapsed tuples for a group. The integration of ties, *i.e.* tuples with the same order level, in the collapsed result may be specified with a '+' mark after the number, indicating that more than K tuples can be integrated if they are ties with the K^{th} tuple. The default behavior is a collapsed result of strictly one tuple.

A special case is to use the Pareto optimality to express a multi-objective query [4]: the optimizing parameters are not an ordered list, but a set. The keyword PARETO replaces the maximum number of tuples in the collapsed result, as all Pareto-optimal tuples are integrated.

EXAMPLE 11 *Different forms of the* COLLAPSE *clause*
In Figure 15 (using the "camera" virtual table with an additional 'quality' attribute), the first COLLAPSE *clause is the default case: for one group, the collapsed set contains only one tuple that maximizes a quality attribute and, in the case of equality for the first criterion, minimizes the cost attribute. The second clause extracts the three least expensive camera, with the best quality in case of equality. The third one uses the same criteria to extract at least two cameras, but may also include cameras that have the same cost and quality as the second best one. The last clause extracts the Pareto-optimal tuples,* i.e. *the one with the best cost and the one with the best quality: as it may be the same tuple, the collapsed set may contain only one tuple or two tuples. Note that in the last clause, the order of the ordering attributes is not relevant.*

```
COLLAPSE (c.cost,c.photo) INTO bestCamera
  USING c.quality DESC, c.cost ASC
COLLAPSE (c.cost,c.photo) INTO bestCamera[3]
  USING c.cost ASC, c.quality DESC
COLLAPSE (c.cost,c.photo) INTO bestCamera[2+]
  USING c.cost ASC, c.quality DESC
COLLAPSE (c.cost,c.photo) INTO bestCamera[PARETO]
  USING quality DESC, c.cost ASC
```

Figure 15: Example of COLLAPSE clauses using two ordering attributes

Although we present it in the context of AOCQs to choose the optimal service(s) to be called for a given event, this clause can be applied to other cases, in particular in non-continuous query, *e.g.* in multi-objective query processing [4] or to declaratively define complex aggregations like in [9, 2].

5 Implementation

Continuous query processing techniques are inspired from standard query processing techniques [16]. However, the introduction of the notion of time impacts on the whole conception.

We propose an architecture of an AOCQ-enabled DSMS. We choose to build our AOCQ processor prototype on top of an open-source DSMS: STREAM [3], whose prototype is developed at Stanford University. We explain the integration of additional functionalities in order to handle some AOCQ concepts, and describe first experimental results from our prototype.

5.1 AOCQ Processor Architecture

The architecture of the AOCQ processor is composed of six main modules, as shown in Figure 16. Query analysis is performed first by the query parser, and then by the query optimizer. The query optimizer checks the query semantics with the metadata catalog, and produces a continuous query evaluation plan. The query plan manager is responsible for the simultaneous execution of all produced plans: it schedules the different query operators in a (near) real-time fashion. The data source manager provides access to the data sources and handles function calls via the service interface manager. This architecture is obviously compliant with the STREAM architecture described in the next section.

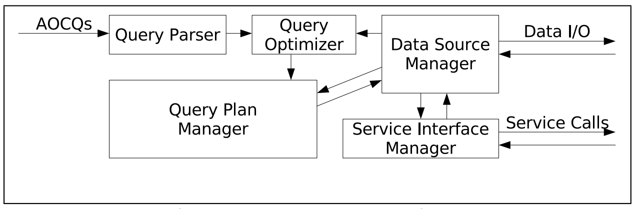

Figure 16: Architecture of the AOCQ processor

5.2 The STREAM Prototype

STREAM provides support for “a large class of declarative continuous queries over continuous streams and traditional stored data sets” [3]. 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.

5.3 Implementation of new Operators

In order to handle AOCQs, we extend the STREAM prototype to integrate the COLLAPSE clause and the notion of binding patterns. For the time being, only limited support for binding patterns has been integrated in the implementation.

The COLLAPSE clause is integrated as a sort of polymorphic aggregation function: it can accept any type and number of input parameters, and its output parameters follow the same schema as the input parameters. We limit the COLLAPSE clause to the default case returning only one tuple, and with only one ordering attribute. Its implementation implies a modification of the analysis of the SELECT clause, and some impacts on the execution of the aggregation operator in query plans.

5.4 Experimentation

We choose to experiment the night surveillance scenario described throughout the paper. The AOCQ is represented in Figure 17. Two tables and two virtual tables have been defined to represent the environment (cf. Example 5). The window specification '[now]' indicates that a tuple from the "sensor" table will not be joined with tuples inserted at a later timestamp in other tables.

```
SELECT s.TIMESTAMP, s.id, p.id
       best.id, best.cost, best.photo,
       best.result
FROM sensor s [now], camera c, phone p, sendMMS
WHERE s.accel_x > 500
  AND s.location = c.location
  AND c.coverage
  AND sendMMS.image = camera.photo
  AND sendMMS.phone_no = p.number
GROUP BY s.TIMESTAMP, s.id, p.id
COLLAPSE (c.cost, c.id, c.photo,
          sendMMS.result) INTO best
  USING c.cost ASC
```

Figure 17: AOCQ for the night surveillance scenario

In order to test the query, test data have been generated for the two tables "sensor" and "phone": 10000 random tuples in "sensor" with a timestamp between 1 and 9999 indicating a 'accel_x' value between 300 and 900 and a location label between 26 possibilities ('a' to 'z'), and 6 tuples in "phone" representing 6 administrators receiving the photos.

50 cameras have been simulated in the virtual table "camera": for each camera, the virtual tuple is represented by one tuple for each location, so that the predicate 's.location = c.location' will select one tuple, and with a random value for the 'coverage' boolean (a location is covered by at least one camera, one camera covers around 20% of the locations) and the 'cost' value. The virtual table "sendMMS" contains one tuple: to simulate the virtual tuple, we simply discard the predicates related to this table.

In order to monitor more closely the query in Figure 17, it is divided into one sub-query joining the tables "sensor" and "camera", producing a stream of tuples representing all possibilities to handle the events, and one main query selecting the optimal tuple by joining the previous stream with the tables "phone" and "sendMMS", and by applying the COLLAPSE clause.

As a result, the sub-query generates a stream of around 76000 tuples. Without the COLLAPSE clause, the main query result set contains more than 450000 tuples, whereas the COLLAPSE clause reduces this number to around 45000, *i.e.* by a factor of 10.

Along with the predictable saving in the number of tuples, this example shows the power of expression of AOCQs: the optimizing criteria being explicitly expressed as the 'cost' attribute of the camera, it can be declaratively changed in the query definition thanks to the COLLAPSE clause. Additional experiments have been scheduled to assess the validity of our prototype.

6 Conclusion

In this paper, we have presented our framework for *Action-Oriented Continuous Queries* (AOCQs) that allows to build queries over relations, streams and services. It is built on top of the CQL specifications [3] that manage streams and relations.

The AOCQ 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. At the query plan level, a dependent join operator realizes a binding pattern. 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 call. 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 builds an implicit table that contains the top-K tuples from a group of tuples according to a given ordering. The COLLAPSE clause intends to replace and augment the procedural and ad hoc user-defined aggregates that are available today in DBMS.

We have also presented first implementation and experimentation results of the COLLAPSE clause on top of the STREAM prototype [3]. For the time being, the prototype includes a mechanism to identify virtual attributes so as to insert dependent joins, invocation operators and binding operators in the execution plan of queries.

This first implementation and experimentation has presented the COLLAPSE clause used in the running example of our article. In future work, we plan to implement invocation and binding operators within STREAM and to develop a benchmark on real data sets and real services.

References

[1] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR 2005, Proceedings of Second Biennial Conference on Innovative Data Systems Research*, 2005.

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

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[4] W.-T. Balke and U. Güntzer. Multi-objective Query Processing for Database Systems. In *VLDB'2004: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 936–947, 2004.

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

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

[7] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. EasyLiving: Technologies for intelligent environments. In *HUC 2000, Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, pages 12–29, 2000.

[8] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR 2003, Proceedings of the First*

Biennial Conference on Innovative Data Systems Research, 2003.

[9] D. Chatziantoniou. The PanQ tool and EMF SQL for Complex Data Management. In *KDD'99: Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 420–424, 1999.

[10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[11] M. Cherniack et al. Scalable Distributed Stream Processing. In *CIDR 2003, Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

[12] L. Ding and E. A. Rundensteiner. Evaluating Window Joins over Punctuated Streams. In *CIKM'04, Proceedings of the 13th ACM international conference on Information and Knowledge Management*, pages 98–107, 2004.

[13] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[14] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *SIGMOD'99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 311–322, 1999.

[15] M. J. Franklin et al. Design Considerations for High Fan-In Systems: The HiFi Approach. In *CIDR 2005, Proceedings of Second Biennial Conference on Innovative Data Systems Research*, 2005.

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

[17] D. Garlan et al. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

[18] R. Goldman and J. Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 285–296, 2000.

[19] R. Grimm et al. System Support for Pervasive Applications. *ACM Transactions on Computer Systems*, 22(4):421–486, November 2004.

[20] K. Henricksen and J. Indulska. A Software Engineering Framework for Context-Aware Pervasive Computing. In *PerCom'04, Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, page 77, 2004.

[21] C.-E. Pigeot, Y. Gripay, M. Scuturici, and J.-M. Pierson. Context-Sensitive Security Framework for Pervasive Environments. In *ECUMN'07: Fourth European Conference on Universal Multiservice Networks*, pages 391–400, 2007.

[22] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query Optimization over Web Services. In *VLDB 2006, Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 355–366, 2006.

[23] F. Tian and D. J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *VLDB 2003, Proceedings of the 29th International Conference on Very Large Data Bases*, pages 333–344, 2003.

[24] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, September 1991.

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

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

[27] F. Zhu, M. Mutka, and L. Ni. Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, 4(4):81–90, 2005.