A Simple (yet Powerful) Algebra for Pervasive Environments

Yann Gripay
Université de Lyon, CNRS
INSA-Lyon, LIRIS, UMR5205
7 avenue Jean Capelle,
F-69621, Villeurbanne, France
yann.gripay@liris.cnrs.fr

Frédérique Laforest
Université de Lyon, CNRS
INSA-Lyon, LIRIS, UMR5205
7 avenue Jean Capelle,
F-69621, Villeurbanne, France
frederique.laforest@
liris.cnrs.fr

Jean-Marc Petit
Université de Lyon, CNRS
INSA-Lyon, LIRIS, UMR5205
7 avenue Jean Capelle,
F-69621, Villeurbanne, France
jean-marc.petit@
liris.cnrs.fr

ABSTRACT

Querying non-conventional data is recognized as a major issue in new environments and applications such as those occurring in pervasive computing. A key issue is the ability to query data, streams and services in a declarative way. Our overall objective is to make the development of pervasive applications easier through database principles. In this paper, through the notion of virtual attributes and binding patterns, we define a data-centric view of pervasive environments: the classical notion of database is extended to come up with a broader notion, defined as relational pervasive environment, integrating data, streams and active/passive services. Then, the so-called Serena algebra is proposed with operators to homogeneously handle data and services. Moreover, the notion of stream can also be smoothly integrated into this algebra. A prototype of Pervasive Environment Management System has been implemented on which first experiments have been conducted to validate our approach.[1]

Categories and Subject Descriptors

H.2.3 [**Information Systems**]: Database Management—*Languages*

General Terms

Design, Languages

1. INTRODUCTION

Computing environments evolve towards what is called pervasive systems [22]: they tend to be more and more heterogeneous, decentralized and autonomous. On the one hand, personal computers and other handheld devices are largely widespread and take a large part of information systems. On the other hand, data sources may be distributed over large areas through networks that range from a worldwide network like the Internet to local peer-to-peer connections like for sensors.

Querying non-conventional data is recognized as a major issue in new environments and applications such as those occurring in pervasive systems. In such environments, available data sources and functionalities are dynamic and heterogeneous: distributed databases with frequent updates, data streams from logical or physical sensors, and services providing data from sensors or storage units, transforming data or commanding actuators. These data sources and functionalities are however not homogeneously manageable in today's systems, which is an issue when building pervasive applications: imperative programming languages (e.g., C++, Java) are needed to define application behavior, whereas data source access is achieved through database connections with declarative language (e.g., SQL) and distributed functionality invocation through specific network protocols (e.g., JMX, UPnP). *Ad hoc* development combining those techniques is not a suitable long-term solution.

DSMSs (Data Stream Management Systems), extending DBMSs (DataBase Management Systems), provide a homogeneous view and query facilities for both relational data and data streams. Services are a common way to represent distributed functionalities in a computing environment, along with techniques for service discovery and remote invocation [26]. However, services are not yet fully integrated with DBMSs or DSMSs. Major DBMS vendors offer SQL extensions with functions and external functions to link with external components, e.g., distributed services. Nevertheless, they do not handle the dynamicity of the set of available services, and functions or services are not considered as first-class citizens in the sense that they cannot be combined with traditional data schemas or queried in a homogeneous way with traditional data.

1.1 Contributions

Our overall objective is to make the development of pervasive applications easier through database principles. We have to manage two types of entities: data sources (data, streams) and services (i.e. distributed functionalities).

Our notion of service is kept rather simple in order to ensure tractable solutions even in dynamic environments. For

[1]This work was partially supported by the ANR (French National Research Agency) project Optimacs (ANR-08-SEGI-014, 2008–2011).

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

example, we assume that service invocations terminate (no infinite behavior) to avoid termination and recursion problems, e.g., see [4]. A service method can be tagged as active or passive to reflect its impact, or absence of impact, on the environment, which is a strong requirement in the context of pervasive environments. For example, the invocation of a service sending a SMS message to someone has an impact on the environment, i.e. once received by the person, the SMS can not be "canceled". Such a service is referred to as an active service in the rest of this paper. On the opposite, a service returning the current temperature value from a sensor can be tagged as a passive service. This notion of active/passive service turns out to be powerful enough to capture most of requirements in pervasive environment.

We propose a framework that defines a data-centric view of pervasive environments: the classical notion of database is extended to come up with a broader notion, defined as relational pervasive environment, integrating data, streams and active/passive services.

In this paper, we detail the key issue of our framework: in order to integrate distributed functionalities, we decouple their declaration as prototypes of methods and their implementation as services; at the metadata level, we integrate the declarative part into data schemas with the help of two notions: *virtual attributes* and *binding patterns*; the implementation part is integrated at the tuple level with data representing references to services. We propose a formal definition of a pervasive environment as a set of extended relations, or X-Relations, and we define a service-enabled algebra over pervasive environments based on the relational algebra, the **Serena** algebra (**Ser**vice-**ena**bled algebra). This model allows to define service-oriented queries in a declarative fashion, whereas implementation issues like service discovery and remote invocation are handled at runtime.

Our framework also combines this integration of services with continuous query techniques (e.g., [23]) over relations and data streams. Continuous aspects are important for pervasive applications and essential for the expressiveness of our framework, but have nevertheless little impact on the definition of the algebra. In this paper, due to space limitations, we only sketch how we realize this combination, by extending our formal definition of a pervasive environment as a set of eXtended Dynamic relations, or XD-Relations, that homogeneously represents relations and data streams extended with services, and by extending the Serena algebra to define query operators over such XD-Relations. The definition of a SQL-like language based on the Serena algebra, namely the Serena SQL, is also not tackled in this paper although it is part of our framework.

1.2 Motivating Example

The motivating example is inspired by the night surveillance scenario presented in the Aorta project [24]. It illustrates the need for the integration of services from a dynamic environment in a database-like framework and for associated declarative languages.

Our temperature surveillance scenario considers a building containing temperature sensors and network cameras. The surveillance consists of controlling temperatures provided by those sensors to trigger a photo of the location of the involved sensors when the temperature (or the mean temperature) exceeds a threshold, and to send it to the administrators via e-mail, instant message or SMS. Photos are taken by the network cameras.

In order to express this behavior in a declarative way, the environment can be described using database principles, as a set of extended relations and data streams.

The first table is a list of temperature sensors, with their location, and a virtual attribute *temperature* ('∗' denotes its absence of value). The value for the virtual attribute *temperature* is retrieved, at query evaluation time, by queries involving invocations on the services identified by the *sensor* attribute (e.g., *sensor*01, *sensor*22). In this application, the application designer is likely to tag those services as passive.

sensor	*location*	*temperature*
sensor01	corridor	∗
sensor06	office	∗
sensor07	office	∗
sensor22	roof	∗

This table may be updated by a *dynamic service discovery mechanism*, and thus represent the set of all available services providing localized temperature information. New sensors could then be automatically discovered and added to the table, while sensors that are deactivated (or failing) could be also automatically removed from the table.

From this table, a one-shot query can retrieve temperatures, or compute a mean temperature, for a given location. A continuous query can build a temperature stream (with the location) from all the available sensors, for example to feed a monitoring application drawing temperature graphs in real-time.

The second table is an electronic contact list that enables queries to send messages. A query can provide a value for the virtual attributes *text* (the message content) and then retrieve the value of the virtual attribute *sent* (giving the sending result), involving invocations on the active services identified by the attribute *messenger* (e.g., *email*, *jabber*).

name	*address*	*text*	*messenger*	*sent*
Nicolas	nicolas@elysee.fr	∗	email	∗
Carla	carla@elysee.fr	∗	email	∗
François	francois@im.gouv.fr	∗	jabber	∗

A one-shot query can simply send a message to some contacts. A continuous query can combine this table with the previous temperature stream to send alerts when some temperatures exceeds a given value.

The whole temperature surveillance scenario can then be expressed by a continuous query combining the two previous tables with two additional tables: one for the cameras (like for the temperature sensors), and one indicating which contact is in charge of which location (e.g., Carla wants to know when the temperature in Nicolas's office exceeds 28℃).

Through this representation, the different operations of the scenario (resource discovery, service invocations) can be expressed in terms of operations on tables (updating tables, retrieving values of virtual attributes, combining tables, etc.). Pervasive applications like this scenario can then be expressed using declarative one-shot or continuous queries, and query optimization techniques can be applied to optimize the execution of those applications.

1.3 Paper Organization

The rest of the paper is organized as follows. In Section 2, we define a formal model for pervasive environments as an

extended database. In Section 3, we define the Serena algebra over such pervasive environments. In Section 4, we sketch the integration of continuous aspects into our framework. In Section 5, we introduce our PEMS (Pervasive Environment Management System) prototype and some experiments we have conducted so far. In Section 6, we position our research problem within the related work. We then conclude and discuss some perspectives in Section 7.

2. MODELING OF PERVASIVE ENVIRONMENTS

In this section, we describe our generic representation of the pervasive environment. We explain our choices to represent distributed functionalities as prototypes and services in Section 2.1. Prototypes and services can be integrated within standard relations with virtual attributes and binding patterns. First intuitions are given in Section 2.2, and then a formalization of a new data model for pervasive environments is proposed in Section 2.3.

2.1 Prototypes and Services

As distributed functionalities may appear or disappear dynamically in pervasive environments, applications can be defined using abstract functionalities: they are dynamically linked to available implementations at runtime. We choose to represent the distributed functionalities in a way that decouples the declaration of those functionalities (e.g., sending a message to someone) and their implementations (e.g., an e-mail sender, an instant message sender).

We represent the *declaration* of the *distributed functionalities* as **prototypes**. In order to stay close to the relational model, input parameters and output parameters are defined by two relation schemas. For prototype invocation, input parameters take the form of a relation over the input schema (generally with only one tuple) and the invocation result is a relation over the output schema (0, 1 or several tuples).

We denote the *implementations* of those *distributed functionalities* by methods provided by **services**. As those methods correspond to some prototypes, we consider that services **implement** those prototypes. As only prototypes implemented by services are pertinent for the model, methods provided by services may remain implicit and can be safely hidden. The implementation of prototypes are assumed to terminate and, in order to keep the model simple, recursion is not allowed [4].

From a system point of view, the notion of method is only handled by the service discovery and invocation mechanism. Available remote services are discovered and registered, so that a prototype can be invoked on any registered service that implements this prototype: the corresponding method provided by this service is transparently called.

As invocations of prototype on services can have an impact on the physical environment, e.g., invoking a prototype that sends a message, we need to consider two categories of prototypes: **active prototypes** and **passive prototypes**. Active prototypes are prototypes having a side effect on the physical environment that can not be neglected. On the opposite, the impact of passive prototypes is non-existent or can be neglected, like reading sensor data.

EXAMPLE 1. *From the temperature surveillance scenario, 4 prototypes and 9 services that implement them are presented in a pseudo-DDL format, in Table 1. Those prototypes correspond to some functionalities from the environment: sending a message to someone, checking if a photo of an area can be taken (indicating the expected delay and photo quality), taking a photo, and retrieving temperature values. The prototype sendMessage is an active prototype, whereas the three others are passive. The 9 services are identified by their service references, i.e. their identifiers: email, jabber, camera01, sensor06...*

Table 1: Example of Prototypes and Services

```
PROTOTYPE sendMessage( address STRING, text STRING ) :
  (sent BOOLEAN) ACTIVE;

PROTOTYPE checkPhoto( area STRING ) :
  ( quality INTEGER, delay REAL );

PROTOTYPE takePhoto( area STRING, quality INTEGER ) :
  ( photo BLOB );

PROTOTYPE getTemperature(  ) : ( temperature REAL );

SERVICE email    IMPLEMENTS sendMessage;
SERVICE jabber   IMPLEMENTS sendMessage;
SERVICE camera01 IMPLEMENTS checkPhoto, takePhoto;
SERVICE camera02 IMPLEMENTS checkPhoto, takePhoto;
SERVICE webcam17 IMPLEMENTS checkPhoto, takePhoto;
SERVICE sensor01 IMPLEMENTS getTemperature;
SERVICE sensor06 IMPLEMENTS getTemperature;
SERVICE sensor07 IMPLEMENTS getTemperature;
SERVICE sensor22 IMPLEMENTS getTemperature;
```

2.2 Extending Relations with Services

In this section, we keep the presentation informal: the formalization is given in the next section. We propose to integrate prototypes and services at two levels. At the metadata level, prototypes are integrated into relation schemas extended with two notions: **virtual attributes** and **binding patterns**. At the data level, **service references** are considered like any other data values.

The notions of virtual attributes and binding patterns turn out to be critical in our modeling. Virtual attributes offer the opportunity to pose queries over data and services, the retrieval of their value at query execution time being defined through binding patterns.

We consider *service references* to be classical data values identifying services (e.g., integer values, or string values like in Example 1). Attributes representing service references are not different from other attributes and contain, at the tuple level, so-called service references.

In order to represent potential interactions with services, we extend relation schemas with *virtual attributes*. Virtual attributes are defined at the relation schema level only and do not have a value at the tuple level. They can be transformed into *real attributes*, i.e. non-virtual attributes, through some query operators of our algebra (defined in Section 3): their value is then set at query execution time.

Extending a relation schema with virtual attributes does not influence the tuple representation for its relations. Virtual attributes only represent potential attributes that can be used by queries.

Binding patterns are the relationship between service references, virtual attributes and prototypes. A binding pattern is associated with a relation schema and defines which prototype to invoke on services in order to retrieve values for one or more virtual attributes. It also specifies the real

attribute representing service references. The prototype indicates which attributes from the relation schema are input parameters and output parameters. Input parameters are real or virtual attributes, whereas output parameters are always virtual attributes. A binding pattern is said to be **active** if its associated prototype is active, **passive** otherwise. Like for virtual attributes, associating binding patterns with a relation schema does not influence the tuple representation for its relations. Binding patterns represent a potential way to provide values for virtual attributes that can be used by queries.

Consequently, a relation over an extended relation schema, i.e. a relation schema extended with virtual attributes and associated with binding patterns, is called an **extended relation**, or **X-Relation**. Therefore, a **relational pervasive environment** is seen as a set of X-Relations.

Example 2 (Relational Pervasive Environment). *Continuing Example 1, the relational pervasive environment for the temperature surveillance scenario is represented in Table 2 using a pseudo-DDL.*

Table 2: Description of X-Relations from the Relational Pervasive Environment for the Temperature Surveillance Scenario

```
EXTENDED RELATION contacts (
  name      STRING,
  address   STRING,
  text      STRING   VIRTUAL,
  messenger SERVICE,
  sent      BOOLEAN  VIRTUAL
)
USING BINDING PATTERNS (
  sendMessage[messenger] ( address, text ) : ( sent )
);

EXTENDED RELATION cameras (
  camera  SERVICE,
  area    STRING,
  quality INTEGER  VIRTUAL,
  delay   REAL     VIRTUAL,
  photo   BLOB     VIRTUAL
)
USING BINDING PATTERNS (
  checkPhoto[camera] ( area ) : ( quality, delay ),
  takePhoto[camera] ( area, quality ) : ( photo )
);
```

2.3 A New Data Model

In this section, we borrow notations from [18]. Before introducing virtual attributes and binding patterns, we first formally define relations, prototypes and services. We then define extended relations with virtual attributes and binding patterns, to build the notion of relational pervasive environment.

2.3.1 Preliminaries

Two disjoint countable infinite sets are defined: $\mathcal{D}$ for constants and $\mathcal{A}$ for attribute names, with $\mathcal{D} \cap \mathcal{A} = \emptyset$. For a given relation schema associated with the relation symbol R, we denote by:

- $type(R)$ the number of attributes in R,
- $att_R : \{1, ..., type(R)\} \rightarrow \mathcal{A}$ a one-to-one (i.e. injective) function mapping numbers to attributes in R,
- $schema(R)$ the set of attributes in R, i.e. $\{att_R(1), ..., att_R(type(R))\} \subseteq \mathcal{A}$.

A *tuple over a relation schema* R is an element of $\mathcal{D}^{type(R)}$, and a *relation over* R is a finite set of tuples over R.

Tuples from relations can be projected onto an attribute A_i or, by generalization, onto a group of attributes X. Let R be a relation schema, r a relation over R, t a tuple from r, $A_i \in schema(R)$ with $A_i = att_R(i)$ and $X = \{att_R(i_1), ..., att_R(i_n)\} \subseteq schema(R)$:

- the projection of a tuple t onto A_i, noted $t[A_i]$, is the i^{th} coordinate of t, i.e. $t(i)$,
- the projection of a tuple t onto X, noted $t[X]$, is defined by $t[X] = \langle t(i_1), ..., t(i_n) \rangle$.

Two other countable infinite sets are defined: Ψ for prototypes and Ω for services, with $\mathcal{D}$, $\mathcal{A}$, Ψ and Ω mutually disjoint.

Prototypes are defined by two relation schemas, one for the input and the other for the output. Those schemas are supposed to be disjoint and the output relation schema has to be non-empty. Let $\psi \in \Psi$ be a prototype, we denote by:

- $Input_\psi$ the input relation schema of prototype ψ,
- $Output_\psi$ the output relation schema of prototype ψ, with $schema(Output_\psi) \neq \emptyset$,
- $schema(Input_\psi) \cap schema(Output_\psi) = \emptyset$,
- $active(\psi)$ a predicate returning true if ψ is active.

Services are defined by the finite set of prototypes they implement. A service is associated with a constant that is its service reference. Let $\omega \in \Omega$ be a service, we denote by:

- $prototypes(\omega) \subset \Psi$ the finite set of prototypes implemented by service ω,
- $id(\omega) \in \mathcal{D}$ the service reference of ω.

Example 3 (Prototypes and Services). *For the temperature surveillance scenario, we define 4 prototypes and 9 services. The 4 prototypes are sendMessage, checkPhoto, takePhoto and getTemperature. For example, the input and output relation schemas associated with sendMessage are* $Input_{sendMessage}$ *and* $Output_{sendMessage}$*, with:*

- $schema(Input_{sendMessage}) = \{address, text\}$,
- $schema(Output_{sendMessage}) = \{sent\}$.

The 9 services from Ω *are* ω_1, ω_2, ω_3, ..., ω_8 *and* ω_9*, corresponding to the following service references: email, jabber, camera01, ..., sensor07 and sensor22 (cf. Example 1). Each service implements some prototypes. For* ω_1 *and* ω_3*, we note:*

- $id(\omega_1) = email$,
- $prototypes(\omega_1) = \{sendMessage\}$,
- $id(\omega_3) = camera01$,
- $prototypes(\omega_3) = \{checkPhoto, takePhoto\}$.

A prototype invocation on a service can now be formally represented as a function associated with each prototype. For a given prototype, this function takes two parameters: a constant, that should be a service reference, and a tuple over the prototype input schema, providing the input parameters; and returns a relation over the prototype output schema, representing the output parameters.

Definition 1 (Invocation Function). *For each prototype $\psi \in \Psi$, an* invocation function *$invoke_\psi$ is defined:*

$$invoke_\psi : \begin{array}{ccc} (\mathcal{D}, \mathcal{D}^{type(Input_\psi)}) & \rightarrow & \mathcal{P}(\mathcal{D}^{type(Output_\psi)}) \\ (s,t) & \mapsto & r \end{array}$$

This function represents an invocation of prototype ψ on service ω referenced by $s = id(\omega)$. The input parameters are defined by the tuple t over $Input_\psi$. The invocation results are represented by a relation r over $Output_\psi$, i.e. a set of tuples containing the output parameters.

2.3.2 Extended Relations

With these definitions for relations, prototypes, services and service references, we now propose a formal definition of extended relations with virtual attributes and binding patterns. An extended relation is basically defined like a relation. However, its schema is partitioned between a real schema containing real attributes and a virtual schema containing virtual attributes. An extended relation schema is also associated with a finite set of binding patterns: each binding pattern specifies a prototype and an attribute from the real schema representing a service reference. Tuples from an extended relation are defined only over the real schema, i.e. the subset of real attributes, as virtual attributes do not have a value. We then define the notion of extended relation schema, and the notion of extended relation over an extended relation schema.

Definition 2 (Extended Relation Schema). *An* extended relation schema *is an extended relation symbol R associated with:*

- *$type(R)$ the number of attributes in R,*
- *$att_R : \{1, ..., type(R)\} \mapsto \mathcal{A}$ a one-to-one (i.e. injective) function mapping numbers to attributes in R,*
- *$schema(R)$ the set of attributes in R, i.e. $\{att_R(1), ..., att_R(type(R))\} \subseteq \mathcal{A}$,*
- *$\{realSchema(R), virtualSchema(R)\}$ a partition of $schema(R)$ with:*
  - *$realSchema(R)$ the real schema, i.e. the subset of real attributes,*
  - *$virtualSchema(R)$ the virtual schema, i.e. the subset of virtual attributes;*
- *$BP(R) \subset (\Psi \times \mathcal{A})$ a finite set of binding patterns associated with R, where $bp = \langle prototype_{bp}, service_{bp} \rangle \in BP(R)$ with:*
  - *$prototype_{bp} \in \Psi$ the prototype associated with the binding pattern,*
  - *$service_{bp} \in realSchema(R)$ a real attribute from the schema used as a service reference attribute for the binding pattern,*

  constrained by the following restrictions:
  - *$schema(Input_{prototype_{bp}}) \subset schema(R)$,*
  - *$schema(Output_{prototype_{bp}}) \subseteq virtualSchema(R)$.*

We denote by $active(bp)$, with $bp \in BP(R)$, a predicate that returns true if the binding pattern bp is active, i.e. if its associated prototype is active: $active(bp) = active(prototype_{bp})$.

Definition 3 (Extended Relation). *A* tuple over an extended relation schema *R is an element of $\mathcal{D}^{|realSchema(R)|}$. An* extended relation over *R (or X-Relation over R) is a finite set of tuples over R.*

Tuples from X-Relations can be projected onto an attribute A_i or, by generalization, onto a group of attributes X. However, as virtual attributes do not have a value, tuples can only be projected onto real attributes: the coordinate corresponding to the i^{th} attribute is the j^{th} coordinate where j is the number of real attributes in the partial schema $\{att_R(1), ..., att_R(i)\}$. We denote this number by $\delta_R(i)$. To keep the generalization, tuples can be projected onto a set of real attributes $X \subseteq realSchema(R)$.

Definition 4 (Projection of a Tuple). *The* projection of a tuple *t, from an X-Relation r over an extended relation schema R, onto an attribute $A_i \in realSchema(R)$ with $A_i = att_R(i)$ in $schema(R)$, noted $t[A_i]$, is the $\delta_R(i)^{th}$ coordinate of t, i.e. $t(\delta_R(i))$, with $\delta_R(i)$ the number of real attributes in $\{att_R(1), ..., att_R(i)\}$, i.e. $\delta_R(i) = |realSchema(R) \cap \{att_R(1), ..., att_R(i)\}|$.*

The projection of a tuple t onto $X = \{att_R(i_1), ..., att_R(i_n)\} \subseteq realSchema(R)$, noted $t[X]$, is $t[X] = \langle t(\delta_R(i_1)), ..., t(\delta_R(i_n)) \rangle$.

Example 4 (Extended Relation). *The electronic contact list from the temperature surveillance scenario can be modeled by an extended relation schema Contact with the following attributes:*
$schema(Contact) = \{name, address, text, messenger, sent\}$, $realSchema(Contact) = \{name, address, messenger\}$ and $virtualSchema(Contact) = \{text, sent\}$.

messenger is a service reference attribute. text and sent are two virtual attributes representing the text to be sent and the result of the sending. It is associated with one binding pattern $\langle sendMessage, messenger \rangle$. Let contacts be an X-Relation over Contact, it can be represented in the following table, where '' denotes the absence of value for virtual attributes:*

name	*address*	*text*	*messenger*	*sent*
Nicolas	*nicolas@elysee.fr*	*	*email*	*
Carla	*carla@elysee.fr*	*	*email*	*
François	*francois@im.gouv.fr*	*	*jabber*	*

This relation contains three tuples, defined as elements of $\mathcal{D}^{|realSchema(Contact)|} = \mathcal{D}^3$.

Let $t \in \mathcal{D}^3$ be the first tuple from the table:

- *$t = \langle Nicolas, nicolas@elysee.fr, email \rangle$*
- *$t[messenger] = t[att_{Contact}(4)] = \langle t(\delta_{Contact}(4)) \rangle = \langle t(3) \rangle = \langle email \rangle$*
- *$t[address, messenger] = t[att_{Contact}(2), att_{Contact}(4)] = \langle t(\delta_{Contact}(2)), t(\delta_{Contact}(4)) \rangle = \langle t(2), t(3) \rangle = \langle nicolas@elysee.fr, email \rangle$*

With this formalization, standard relations are a special case of extended relations that only have real attributes, and then no virtual attributes or associated binding patterns. Extended relations are a mean to represent some parts of a pervasive computing system. We propose to define the notion of relational pervasive environment representing a set of extended relations, similarly to the notion of database representing a set of relations. For the sake of simplicity, we keep the Universal Relation Schema Assumption (URSA) stating that if an attribute appears in several relation schemas, then this attribute represents the same data.

Definition 5 (Relational Pervasive Env. Schema). *A relational pervasive environment schema P is a finite set $P = \{R_1, ..., R_n\}$, with R_i an extended relation schema. We denote by $schema(P)$ the set of all attributes associated with the extended relation schemas in P, i.e. $schema(P) = \bigcup_{R_i \in P} schema(R_i)$.*

Definition 6 (Relational Pervasive Environment). *A relational pervasive environment over a relational pervasive environment schema $P = \{R_1, ..., R_n\}$ is a set $p = \{r_1, ..., r_n\}$, with $r_i \in p$ an extended relation over $R_i \in P$.*

3. ALGEBRA FOR SERVICE-ORIENTED QUERIES

We propose to define an algebra over this formalization to allow the expression of queries over a relational pervasive environment. We call this algebra **Serena**, standing for **Ser**vice-**ena**bled algebra.

3.1 Query Operators

In this section, we redefine set operators and relational operators. We also define new operators that handle virtual attributes, called realization operators.

3.1.1 Set Operators

The set operators union, intersection and difference can be applied over two X-Relations associated with the same schema. The resulting X-Relation is defined over the same schema. Their definitions remain similar with definitions over standard relations. For example, let r_1 and r_2 be two X-Relations over R, $r_1 \cup r_2 = \{t \mid t \in r_1 \vee t \in r_2\}$.

3.1.2 Relational Operators

Standard relational operators are defined over one or two standard relations. We extend their definitions over one or two X-Relations and study in particular the schema of the resulting X-Relations, because modification or disappearance of some real or virtual attributes can modify or invalidate binding patterns using those attributes.

The *projection* operator (cf. Table 3 (a)) reduces the schema of an X-Relation, thus its real and virtual schemas. The resulting relation is associated with binding patterns from the initial X-Relation that remain valid, i.e. binding patterns with their service reference attribute, input and output attributes that are still in the schema of the resulting X-Relation.

The *selection* operator (cf. Table 3 (b)) does not modify the schema of the X-Relation. However, selection formulas can only apply on attributes from the real schema, as virtual attributes do not have a value. Apart from this restriction, we keep the standard definition and notation for the logical implication [18] ($t \models F$, with t a tuple and F a selection formula over a relation schema R).

The *renaming* operator (cf. Table 3 (c)) replaces an attribute from the schema by another attribute. This operation does not modify the real or virtual status of the renamed attribute. The renaming also impacts on the binding patterns using this attribute.

The *natural join* operator (cf. Table 3 (d)) joins two X-Relations, the join attributes being given by the intersection of the two schemas. If a given join attribute is a real (resp. virtual) attribute in both operands, it remains real (resp. virtual) in the resulting X-Relation. However, if it is real in one operand and virtual in the other one, it becomes real in the resulting X-Relation (i.e. it is an implicit realization of the virtual attribute, see Section 3.1.3). As tuples can not be projected onto virtual attributes, only join attributes that are real in both operands imply a join predicate. If all join attributes are virtual in at least one operand, the join is equivalent, at the tuple level, to a Cartesian product.

The binding patterns for the resulting relation are the union of the binding patterns from both operands, but some may be eliminated if they use output attributes that have become real attributes with the join.

3.1.3 Realization Operators

We introduce two new operators, referred to as *realization operators*. They allow to transform virtual attributes into real attributes. We call this transformation "realization" of virtual attributes. The reverse transformation is not possible. Realized attributes are given a value by the operators, either directly (assignment operator) or using a binding pattern (invocation operator). An implicit realization can also occur in a natural join when a join attribute is real in one operand and virtual in the other one: the virtual attribute from the operand becomes a real attribute in the resulting X-Relation.

The *assignment* operator (cf. Table 3 (e)) over a X-Relation is the realization operator for individual virtual attributes. It allows to give a value to one virtual attribute. In a similar way to simple selection formulas, this operator allows to assign either the value of a real attribute from the schema or a constant value. The virtual attribute thus becomes a real attribute in the resulting X-Relation. The binding patterns for the resulting relation are those of the operand, but some may be eliminated if the realized attribute is part of their output attributes, as it is now a real attribute.

The *invocation* operator (cf. Table 3 (f)) over a X-Relation is the realization operator for the output attributes of a binding pattern. It allows to give them a value by invoking the binding pattern, i.e. invoking the associated prototype with some input parameters on a service. However, this operator can only be applied if all the input attributes of the binding pattern are real attributes in the input X-Relation.

Each tuple from the operand leads to an invocation of the binding pattern, which may result in several tuples for output attributes. Each input tuple is duplicated as many times as the invocation has generated output tuples. For each invocation, the input parameters and the service reference are taken from the input tuple.

3.1.4 Queries

The notion of query over a relational pervasive environment can now be defined as a composition of operators over a set of X-Relations.

Definition 7 (Query over a Relational Perv. Env.). *A query over a relational pervasive environment is a well-formed expression composed of a finite number of Serena algebra operators whose operands are X-Relations.*

Example 5 (Query over X-Relations). *We can express the following queries over X-Relations contacts and cameras (introduced in Example 2, defined in Example 4):*

Table 3: Definition of Relational and Realization Operators over X-Relations

(a)	**Projection**
Input	r an X-Relation over R $Y \subset schema(R)$
Syntax	$s = \pi_Y(r)$
Output	s an X-Relation over S with: - $schema(S) = Y$ - $realSchema(S) = realSchema(R) \cap Y$ - $virtualSchema(S) = virtualSchema(R) \cap Y$ - $BP(S) = \{bp \mid bp \in BP(R) \wedge service_{bp} \in Y \wedge schema(Input_{prototype_{bp}}) \subset Y \wedge schema(Output_{prototype_{bp}}) \subset Y\}$
Tuples	$s = \{t[Y \cap realSchema(R)] \mid t \in r\}$

(b)	**Selection**
Input	r an X-Relation over R F a selection formula over $realSchema(R)$
Syntax	$s = \sigma_F(r)$
Output	s an X-Relation over R
Tuples	$s = \{t \mid t \in r \wedge t \models F\}$

(c)	**Renaming**
Input	r an X-Relation over R $A \in schema(R)$ $B \in \mathcal{A}, B \notin schema(R)$
Syntax	$s = \rho_{A \to B}(r)$
Output	s an X-Relation over S with: - $schema(S) = (schema(R) - \{A\}) \cup \{B\}$ - $realSchema(S) = \begin{cases} (realSchema(R) - \{A\}) \cup \{B\} \text{ if } A \in realSchema(R) \\ realSchema(R) \text{ otherwise} \end{cases}$ - $virtualSchema(S) = \begin{cases} (virtualSchema(R) - \{A\}) \cup \{B\} \text{ if } A \in virtualSchema(R) \\ virtualSchema(R) \text{ otherwise} \end{cases}$ - $BP(S) = \{bp' \mid \exists bp \in BP(R), prototype_{bp'} = prototype_{bp} \wedge$ $((A \neq service_{bp} \wedge service_{bp'} = service_{bp}) \vee (A = service_{bp} \wedge service_{bp'} = B)) \wedge$ $schema(Input_{prototype_{bp}}) \subset (schema(R) - \{A\}) \cup \{B\} \wedge schema(Output_{prototype_{bp}}) \subset (schema(R) - \{A\}) \cup \{B\}\ \}$
Tuples	$s = \{t \mid \exists u \in r, t[realSchema(S) - \{B\}] = u[realSchema(R) - \{A\}] \wedge$ $t[\{B\} \cap realSchema(S)] = u[\{A\} \cap realSchema(R)]\}$

(d)	**Natural Join**
Input	r_1 an X-Relation over R_1 r_2 an X-Relation over R_2
Syntax	$s = r_1 \bowtie r_2$
Output	s an X-Relation over S with: - $schema(S) = schema(R_1) \cup schema(R_2)$ - $realSchema(S) = realSchema(R_1) \cup realSchema(R_2)$ - $virtualSchema(S) = (virtualSchema(R_1) - realSchema(R_2)) \cup (virtualSchema(R_2) - realSchema(R_1))$ - $BP(S) = \{bp \mid bp \in (BP(R_1) \cup BP(R_2)) \wedge schema(Output_{prototype_{bp}}) \subseteq$ $(virtualSchema(R_1) - realSchema(R_2)) \cup (virtualSchema(R_2) - realSchema(R_1))\ \}$
Tuples	$s = \{t \mid \exists t_1 \in r_1, \exists t_2 \in r_2, t[realSchema(R_1)] = t_1 \wedge t[realSchema(R_2)] = t_2\}$

(e)	**Assignment**
Input	r an X-Relation over R $A \in virtualSchema(R)$ $B \in realSchema(R)$ or $a \in \mathcal{D}$
Syntax	$s = \alpha_{A \equiv B}(r)$ or $s = \alpha_{A \equiv a}(r)$
Output	s an X-Relation over S with: - $schema(S) = schema(R)$ - $realSchema(S) = realSchema(R) \cup \{A\}$ - $virtualSchema(S) = virtualSchema(S) - \{A\}$ - $BP(S) = \{bp \mid bp \in BP(R) \ \wedge schema(Output_{prototype_{bp}}) \subseteq (virtualSchema(R) - \{A\})\}$
Tuples	$s = \{t \mid \exists u \in r, t[realSchema(S) - \{A\}] = u[realSchema(R)] \wedge t[A] = u[B]\}$ $s = \{t \mid \exists u \in r, t[realSchema(S) - \{A\}] = u[realSchema(R)] \wedge t[A] = a\}$

(f)	**Invocation**
Input	r an X-Relation over R $bp \in BP(R)$ $schema(Input_{prototype_{bp}}) \subset realSchema(R)$
Syntax	$s = \beta_{bp}(r)$
Output	s an X-Relation over S with: - $schema(S) = schema(R)$ - $realSchema(S) = realSchema(R) \cup schema(Output_{prototype_{bp}})$ - $virtualSchema(S) = virtualSchema(R) - schema(Output_{prototype_{bp}})$ - $BP(S) = \{bp' \mid bp' \in BP(R) \wedge schema(Output_{prototype_{bp'}}) \subseteq (virtualSchema(R) - schema(Output_{prototype_{bp}}))\}$
Tuples	$s = \{t \mid \exists u \in r, t[realSchema(S) - schema(Output_{prototype_{bp}})] = u[realSchema(R)] \wedge$ $t[schema(Output_{prototype_{bp}})] \in invoke_{prototype_{bp}}(u[service_{bp}], u[schema(Input_{prototype_{bp}})])\}$

Table 4: Examples of queries expressed in the Serena algebra

Q_1	$\beta_{\langle sendMessage, messenger\rangle}(\alpha_{text \equiv \text{"Bonjour!"}}(\sigma_{name \neq \text{"Carla"}}(contacts)))$
Q_2	$\pi_{photo}(\beta_{\langle takePhoto, camera\rangle}(\sigma_{quality>5}(\beta_{\langle checkPhoto, camera\rangle}(\sigma_{area=\text{"office"}}(cameras)))))$
Q'_1	$\sigma_{name \neq \text{"Carla"}}(\beta_{\langle sendMessage, messenger\rangle}(\alpha_{text \equiv \text{"Bonjour!"}}(contacts)))$
Q'_2	$\pi_{photo}(\sigma_{quality>5}(\beta_{\langle takePhoto, camera\rangle}(\beta_{\langle checkPhoto, camera\rangle}(\sigma_{area=\text{"office"}}(cameras)))))$
Q_3	$\beta_{\langle sendMessage, messenger\rangle}(\alpha_{text \equiv \text{"Hot!"}}(contacts \bowtie \sigma_{temperature>35.5}(\mathcal{W}_{[1]}(temperatures))))$
Q_4	$\mathcal{S}_{[insertion]}(\beta_{\langle takePhoto, camera\rangle}(\beta_{\langle checkPhoto, camera\rangle}(cameras \bowtie \sigma_{temperature<12.0}(\mathcal{W}_{[1]}(temperatures)))))$

Q_1 *send the message "Bonjour!" to all contacts, except "Carla";*

Q_2 *take photos of area "office" with quality superior to "5".*

The Serena algebra expression for those queries are presented in Table 4.

3.2 Query Equivalence

Without formal semantics, it is hard to prove correctness of query formulations and query optimization is *de facto* limited, operators being often seen as "black boxes". As a matter of fact, logical query optimization is now possible in our setting. In this section, we define query equivalence for service-oriented queries. Three issues need to be addressed in the context of pervasive environments: time-dependence, service determinism and impact of service invocations on the physical environment.

As a pervasive system is a dynamic system, the same service invoked with the same input, but at two different instants in time may lead to two different results (e.g., a service that takes a photo). As a consequence, the same query q over the same relational pervasive environment may lead to different results if q is evaluated at different instants. In order to define query equivalence, we consider a discrete time domain and we assume that query evaluation occurs at a given time instant. As a consequence, all service invocations in a query occur simultaneously, from a theoretical point of view. We also consider that services are deterministic at a given instant, so that the invocation order has no impact on invocation results. For example, a service that returns the number of times it has been invoked should still return the same value for all invocations at a given instant.

In order to reflect the impact of a query on the environment, we define the notion of *action set* induced by a query against a relational pervasive environment as the set of invocations of *active binding patterns* triggered by this query. For instance, consider query Q_1 in Table 4: we want to capture the set of messages sent by the execution of this query.

Definition 8 (Action Set). *An* action *is a 3-tuple $\langle bp, s, t\rangle$ with bp an active binding pattern, s a service reference and t an input data tuple for bp. An* action set *of a query q against a relational pervasive environment p, denoted by $Actions_q(p)$ (or simply $Actions_q$ where p is clear from context), is defined as: $Actions_q(p) = \{\langle bp, s, t\rangle \mid \beta_{bp}(q'(p)) \in q \wedge active(bp) \wedge u \in q'(p) \wedge s = u[service_{bp}] \wedge t = u[schema(Input_{prototype_{bp}})]\}$, where $\beta_{bp}(q'(p)) \in q$ denotes the occurrence of invocation operator β_{bp} in q with subquery q' as its operand.*

Example 6 (Action Sets). *Considering the X-Relation contacts described in Example 4, the action sets for the two similar queries Q_1 and Q'_1 from Table 4 are:*
let $bp_1 = \langle sendMessage, messenger\rangle$,
$Actions_{Q_1} = \{$
$\langle bp_1, email, \langle nicolas@elysee.fr, Bonjour!\rangle\rangle$,
$\langle bp_1, jabber, \langle francois@im.gouv.fr, Bonjour!\rangle\rangle\}$,
$Actions_{Q'_1} = \{$
$\langle bp_1, email, \langle nicolas@elysee.fr, Bonjour!\rangle\rangle$,
$\langle bp_1, email, \langle carla@elysee.fr, Bonjour!\rangle\rangle$,
$\langle bp_1, jabber, \langle francois@im.gouv.fr, Bonjour!\rangle\rangle\}$.

Using this model, the evaluation of a query q over a relational pervasive environment is unambiguously defined. We consider a discrete and ordered time domain T of time instants $\tau \in T$ (in a similar way to CQL [23] for data streams). For a given prototype ψ, the invocation function $invoke_\psi$ is defined at every instant τ for all services ω and all possible inputs t. The evaluation of query q over a relational pervasive environment p occurs at a given instant τ: service invocations, through invocation operators, are defined by the corresponding invocation functions at the given instant.

Definition 9 (Query Equivalence). *Two queries q_1 and q_2 over a relational pervasive environment schema P are said to be* equivalent*, denoted by $q_1 \equiv q_2$,* iff *for any p over P, $q_1(p) = q_2(p)$ and $Actions_{q_1}(p) = Actions_{q_2}(p)$.*

In other words, two equivalent queries over a given relational pervasive environment that are evaluated at the same discrete time instant lead to the same result and the same set of invocations of active binding patterns, although they may imply different invocations of passive binding patterns.

Example 7 (Query Equivalence). *Queries Q_1 and Q'_1 from Table 4 are not equivalent because of their action sets (see Example 6), although their resulting X-Relation should be the same. Queries Q_2 and Q'_2 (also from Table 4) are equivalent, as prototypes takePhoto and checkPhoto are passive: their action sets are then both empty.*

3.3 Query Rewriting

Based on this query equivalence, *rewriting rules* can be applied to query expressed in the Serena algebra. Some well-known rewriting rules of the relational algebra are still pertinent and allow to reorganize the order of relational operators in queries.

Realization operators can also be reorganized, except for invocation operators associated with active binding patterns. The opposition between active and passive binding patterns

Table 5: Rewriting rules with assignment and invocation operators

Operator	**Assignment of virtual attribute A (with real attribute B or constant c)**
Projection	$\pi_L(\alpha_{A\equiv B}(r)) \equiv \alpha_{A\equiv B}(\pi_L(r))$ if $A, B \in L$ $\pi_L(\alpha_{A\equiv c}(r)) \equiv \alpha_{A\equiv c}(\pi_L(r))$ if $A \in L$
Selection	$\sigma_F(\alpha_{A\equiv B}(r)) \equiv \alpha_{A\equiv B}(\sigma_F(r))$ if $A \notin F$ $\sigma_F(\alpha_{A\equiv c}(r)) \equiv \alpha_{A\equiv c}(\sigma_F(r))$ if $A \notin F$
Natural Join	$\alpha_{A\equiv B}(r_1 \bowtie r_2) \equiv \alpha_{A\equiv B}(r_1) \bowtie r_2$ if $A, B \in schema(R_1)$ and $A \notin realSchema(R_2)$ $\alpha_{A\equiv c}(r_1 \bowtie r_2) \equiv \alpha_{A\equiv c}(r_1) \bowtie r_2$ if $A \in schema(R_1)$ and $A \notin realSchema(R_2)$
Operator	**Invocation of *passive* binding pattern bp (i.e. where $prototype_{bp}$ is *passive*)**
Projection	$\pi_L(\beta_{bp}(r)) \equiv \beta_{bp}(\pi_L(r))$ if $not(active(bp))$ and $service_{bp} \in L$ and $schema(Input_{prototype_{bp}}) \subset L$ and $schema(Output_{prototype_{bp}}) \subset L$
Selection	$\sigma_F(\beta_{bp}(r)) \equiv \beta_{bp}(\sigma_F(r))$ if $not(active(bp))$ and $schema(Output_{prototype_{bp}}) \cap F = \emptyset$
Natural Join	$\beta_{bp}(r_1 \bowtie r_2) \equiv \beta_{bp}(r_1) \bowtie r_2$ if $not(active(bp))$ and $bp \in BP(R_1)$ and $schema(Input_{prototype_{bp}}) \subset realSchema(R_1)$ and $schema(Output_{prototype_{bp}}) \cap realSchema(R_2) = \emptyset$

should be taken into account: in a similar way to deterministic UDFs (User-Defined Functions) in standard SQL, only invocation operators with passive binding patterns can be reorganized; active binding patterns limit the possibility of rewriting.

Those rewriting rules are shown in Table 5 for both realization operators, i.e. assignment and invocation operators. Their proof, based on reasoning about sets, are omitted as they are rather simple (see the definition of operators in Table 3). As realization operators modify the real/virtual status of attributes, there are some restrictions in those rules, e.g., with a selection operator, attributes realized by the realization operator should not be part of the selection formula in order to push down the selection operator.

An example of rewriting can be found in Table 4: query Q_2 is a rewritten version of query Q_2' (whereas Q_1 and Q_1' are not equivalent). The choice of tagging binding patterns as active or passive is up to the application developer. Whereas *sendMessage* is surely to be defined as active, *takePhoto* may be considered passive (leading to the rewriting of Q_2' into Q_2) or active (leading to the non-equivalence between Q_2' and Q_2), depending on the objectives of the application and the impacts of executing the services.

4. EXTENSION TO CONTINUOUS ASPECTS

To handle continuous processes, e.g., surveillance, monitoring of data, *etc.*, we propose to extend our initial data model and algebra to take into account data streams and continuous queries in our framework. In this paper, we just sketch the main ideas to show how we have smoothly integrated continuous aspects.

4.1 Extended Relations and Streams

Relations and data streams can be represented in a similar way to CQL [23]: both are defined over a relation schema and represent multisets of tuples for each time instant (considering a discrete time). The difference is that relations are finite multisets whereas data streams are infinite append-only multisets.

Therefore, the notion of eXtended Dynamic relation, or XD-Relation, over an extended relation schema can be defined as a mapping from time instant to multisets of tuples over this schema. An XD-Relation may be finite or infinite. The notion of relational pervasive environment is also extended to represent a set of XD-Relations.

4.2 Algebra

The Serena algebra can be extended over XD-Relations to handle time, in order to define continuous queries over XD-Relations. Previously defined operators can be extended over finite XD-Relations by considering their instantaneous relation, i.e. for each time instant, a finite XD-Relation is like an X-Relation. However, the invocation operator needs to be slightly modified to behave as expected: a binding pattern is actually invoked only for newly inserted tuples, and not for every tuple from the relation at each time instant.

Two additional operators can be defined to handle infinite XD-Relations. The Window operator, denoted by $\mathcal{W}_{[period]}$, computes a finite XD-Relation from an infinite XD-Relation as, for every time instant, the multiset of tuples inserted during the last *period* instants. The Streaming operator, denoted by $\mathcal{S}_{[type]}$, computes an infinite XD-Relation from a finite XD-Relation by inserting, for every time instant, the multiset of tuples that are inserted/deleted/present at this instant (depending on the *type* of the operator: *insertion*, *deletion* or *heartbeat*). As those two operators do not modify the XD-Relation schema, apart from its finite/infinite status, they transparently handle virtual attributes.

Example 8 (Continuous Query). *We consider a relational pervasive environment containing three XD-Relations: contacts and cameras, as finite dynamic versions of previously defined X-Relations; and temperatures, an infinite XD-relation representing a temperature stream from sensors periodically providing temperatures associated with locations (as described in the motivating example in Section 1.2).*

Queries Q_3 and Q_4 in Table 4 model those behaviors:

- *Q_3 when a temperature exceeds 35.5°C, send a message "Hot!" to the contacts,*
- *Q_4 when a temperature goes down below 12.0°C, take a photo of the area.*

In both queries, the window operator with a period of 1 ($\mathcal{W}_{[1]}$) indicates that we are interested in data tuples from the temperatures stream only at the instant where they are inserted. They are not kept in the intermediary XD-Relation for the following instants.

Whereas the result of Q_3 is a finite XD-Relation (its last operator is the invocation operator), the result of Q_4 is an infinite XD-Relation (its last operator is the streaming operator $\mathcal{S}_{[insertion]}$), i.e. a stream of photos.

Besides continuous queries, one-shot queries like Q_1 and Q_2 are still possible over finite XD-Relations: their results are computed once and are not continuously updated.

5. IMPLEMENTATION

In order to validate our approach and conduct some experiments, we have developed a prototype of a Pervasive Environment Management System (PEMS). The role of a PEMS is to manage a relational pervasive environment, with its dynamic data sources and set of services, and to execute continuous queries over this environment. We also have defined a Data Description Language for XD-Relations (the Serena DDL) along with a query language representing Serena algebra expressions (the Serena Algebra Language, not presented in this paper) for continuous queries over XD-Relations.

5.1 Prototype

The whole prototype has been developed in Java using the OSGi framework [19], including UPnP technologies [21] for network issues. The three core modules (Environment Resource Manager, Extended Table Manager, Query Processor) and the distributed modules (Local Environment Resource Managers) are packaged as OSGi bundles. The deployment of the different modules and their interactions are illustrated in Figure 1.

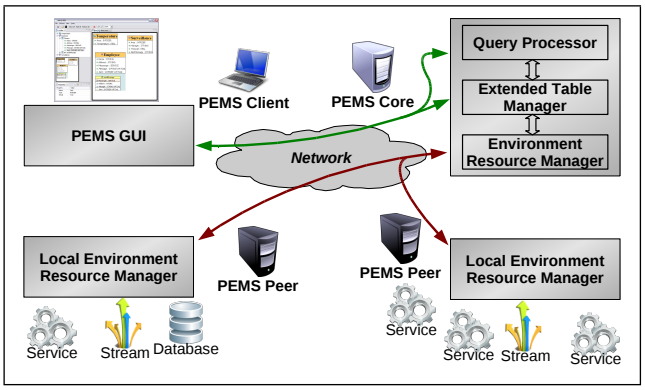

Figure 1: Overview of the PEMS Architecture

The core Environment Resource Manager handles network issues for service discovery and remote invocation, as well as input of data from remote sources (data relations, data streams). It discovers and communicates with Local Environment Resource Managers that are distributed in the network. Services simply register to their Local Environment Resource Manager, and are then transparently available through the core Environment Resource Manager.

The Extended Table Manager allows to define XD-Relations from Serena DDL statements, and to manage their data (insertion and deletion of tuples).

The Query Processor allows to register queries using the Serena Algebra Language and to execute them in a real-time fashion. All relational operators and realization operators have been implemented, as well as the Window and Streaming operators for continuous queries. Service invocations are handled asynchronously by the invocation operator, relying on the core Environment Resource Manager for actual invocations. The Query Processor also handles service discovery queries: it continuously updates some specific XD-Relations so that they represent the set of services (implementing some given prototypes) that are available through the core Environment Resource Manager, like for the XD-Relation *cameras* from the temperature surveillance scenario.

5.2 Experimentation

In order to experiment the temperature surveillance scenario, we have developed an experimental environment composed of several services: physical or simulated temperature sensors (*Thermochron iButton DS1921*), webcams (from *Logitech*), instant messaging server (*Openfire* server from *Jive Software*), (gateway to) SMS gateway (commercial service from *Clickatell*), (gateway to) mail server. Those *distributed functionalities* were wrapped as services and registered to their Local Environment Resource Manager.

An instant messaging client (*Psi*) and a mail client (*Mozilla Thunderbird*) are also used to receive messages, along with a smart phone for SMS. Through the PEMS GUI, XD-Relations have been created on the Extended Table Manager using the Serena DDL, and continuous queries have been registered to the Query Processor using the Serena Algebra Language.

For the temperature surveillance scenario, three XD-Relations have been created: three finite XD-Relations, *cameras*, *surveillance* (indicating who is the "manager" of which area) and *contacts* (with an additional attribute allowing to send a picture with a message); and one infinite XD-Relation *temperatures*. The binding pattern *sendMessage* of *contacts* is active, whereas the binding patterns *checkPhoto* and *takePhoto* of *cameras* are both passive. The continuous query combining these four XD-Relations have been executed: when temperature sensors (physical or simulated) are heated over the threshold specified in *surveillance*, alert messages start to be sent to the "manager" of the associated area, by mail, instant message or SMS. Using an additional service discovery query, new temperature sensors have been dynamically discovered and integrated in the temperature stream without the need to stop the continuous query execution.

We have also experimented another scenario with RSS feeds. A wrapper service transforms RSS feeds into real streams so that a tuple is inserted in the stream when a new item appears in the RSS feed (that is periodically checked). We have tested continuous queries providing the last RSS items containing a given word (e.g., "Obama"), with a one-hour window, from several national and international information websites (french newspapers "Le Monde" and "Le Figaro", and also from "CNN Europe"). The resulting table has been continuously updated, when news of interest appeared and when one-hour-old news expired. Combining this table with the previous finite XD-Relation *contacts*, those news of interest can be sent as messages to a contact.

Those two scenarios (temperature surveillance, RSS feeds) have been successfully tested, showing the feasibility of our declarative approach to simplify the development of pervasive applications, as different kind of data sources and services can be homogeneously queried without much effort. Nevertheless, further experiments need to be conducted to assess the scalability and the robustness of our proposal. Note that in the context of pervasive environment, this is not a trivial issue since, to the best of our knowledge, no benchmark can be used for that purpose.

6. RELATED WORK

Data management in pervasive or ubiquitous computing raises many classical issues in database and distributed system areas. We believe that the closer issues are around pervasive environments, query processing over non conventional data sources such as data streams, data integration and dataspaces.

In this paper, we do not consider contributions related to spatio-temporal data management and mobility issues, even if they are also clearly related to pervasive computing. Nevertheless, they do not enter in the main stream of our contribution and will not be developed further.

6.1 Pervasive Environments

With the development of autonomous devices and location-dependent functionalities, information systems tend to become what Mark Weiser [22] called ubiquitous systems, or pervasive systems. Pervasive systems are based on an abstraction of the distributed functionalities of heterogeneous devices in order to automate some of the possible interactions, e.g., dynamic discovery of devices ([6, 9]), data and application sharing among devices ([13, 15]).

Distributed functionalities can be represented as remote services: dynamic service discovery and remote invocation techniques [26] should be used to handle network issues, like UPnP [21], Web Services, *etc.* As devices may be sensors or actuators [9], 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.

As far as we know, bringing data management principles to pervasive systems as we do, i.e. focusing on service integration issues as well as on data stream issues, has received only minor considerations in the relatively new pervasive system area. Following some of these principles, the PEMS architecture allows to use distributed functionalities without worrying about neither their implementation, nor their invocation.

6.2 Query Processing over Non-Conventional Data Sources

A great number of works have been realized in continuous query definition and processing. Most of works (e.g., [23, 7, 12, 25]) propose an extension of SQL in order to work with both relational databases and data streams, where data streams are represented using relation schemas. Other works tackle continuous querying over distributed XML data sets (e.g., [8]), or use a box representation of query operators [2].

In [24], continuous queries can implicitly interact with devices through an external function call. However, the relationship between functions and devices, as well as the optimization criteria, are not explicit and cannot be declaratively defined. In [17], the cleaning process for data retrieved from physical sensors is defined in a declarative way by a pipeline of continuous queries. It is however only a part of pervasive applications and does not involve services. In [3], the Global Sensor Network allows to define virtual sensors abstracting implementation details, and provides continuous query processing facilities over distributed data streams.

To the best of our knowledge, query processing techniques over non-conventional data have had few impacts on pervasive application developments involving dynamic relations, data streams and services. Through a homogeneous representation for non-conventional data sources, we claim that pervasive application development is indeed possible at the declarative level using service-oriented continuous queries.

6.3 Data Integration and Dataspaces

Data integration has been a long standing theme of research over the past 30 years. Now, the broader notion of dataspaces [11, 16] has appeared to provide base functionality over all data sources and applications, regardless of how integrated they are and without having a full control over the underlying data [11].

In the setting of data integration, the notion of *binding patterns* appears to be quite interesting since they allow to model a restricted 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" [10]. A relation with binding patterns can represent an external data source with limited access patterns [10], or an interface to an infinite data source, e.g., a web site search engine [14]. In a more general way, it can represent a data service, e.g., web services providing data sets, as a virtual relational table [20].

The SQL standard itself supports some forms of access to external functionalities through User-Defined Functions (UDF). UDFs can be scalar functions (returning a single value) or table functions (returning a relation). UDFs are defined in SQL or in another programming language (e.g., C, Java), enabling to access to any external resources. Table functions are a way to implement the notion of virtual tables, however limited to having only one binding pattern determined by the function input parameters. UDFs are also tagged as deterministic or non-deterministic: query rewriting may not change the number of invocations for non-deterministic UDFs.

In a similar way, the ActiveXML language [1] allows to define XML documents containing extensional data, i.e. data that are present in the document, and intensional data, representing service calls that provide data when needed. Intensional data is close to the notion of virtual tables and binding patterns. ActiveXML is also a "framework for distributed XML data management" [5] and defines an algebra to model operations over ActiveXML documents distributed among peers, that enables query optimization.

With our proposition, we aim at contributing in the area of dataspaces through a unified view of data and service spaces mandatory in pervasive environments. Binding patterns play a key role and have been promoted at the schema level to describe distributed functionalities of the environment. Data relations can be extended with virtual attributes, that represent a finer grain of interaction than virtual tables. The declaration of those functionalities are totally decoupled from their implementations that are represented by external services, unlike with standard SQL where specific UDFs should be developed. In comparison with ActiveXML, we do not focus on distributed data management, but on combining data and distributed functionalities in order to build distributed applications for dynamic environments through declarative queries. Their data model [5] is based on trees, while our data model is simpler since based on an extension of the relational model.

7. CONCLUSION

Pervasive systems intend to take advantage of the evolving user environment so as to provide applications adapted to the environment resources. As far as we know, bridging the gap between data management and pervasive applications has not been fully addressed yet. A clear understanding of the interplays between relational data, data streams and services is still lacking and is the major bottleneck toward the declarative definition of pervasive applications.

We have presented a framework that provides a homogeneous view on all available conventional and non-conventional data sources, i.e. databases, data streams and services. The integration of services into relations allows to use a different service for each tuple (e.g., a different messaging service for each contact in a contact list) through the key notions of prototype, service reference, virtual attribute and binding pattern. A formal model of such extended relations has been provided on top of which the **Ser**vice-**ena**bled algebra (Serena algebra) has been proposed. The issue of side effect of service invocations has also been considered to define query equivalence and rewriting rules. A prototype of a Pervasive Environment Management System has been devised, demonstrating the feasibility of our approach.

Future works in this project concern query optimization techniques for service-oriented continuous queries, including a formal definition of *cost models* dedicated to pervasive environments. We are also studying the equivalence of the Serena algebra with some logic-based query languages in order to define a corresponding calculus, and we are investigating a new notion of *streaming binding pattern* to homogeneously integrate in our framework streams provided by services. Furthermore, some existing database extensions, e.g., spatial operators, could also be integrated into the declarative definition of pervasive applications in order to extend their expressiveness.

We also aim at developing a benchmark for pervasive environments to evaluate the performance of "hybrid queries" involving data and services with objective indicators. This benchmark is part of a French National Research Agency (ANR) project called Optimacs, started in December 2008.

8. REFERENCES

[1] ActiveXML. `http://www.activexml.net/`.

[2] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *Proceedings of CIDR'05*, 2005.

[3] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *Proceedings of MDM2007*, pages 198–205. IEEE, 2007.

[4] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *Proceedings of PODS'04*, pages 35–45, 2004.

[5] S. Abiteboul, I. Manolescu, and E. Taropa. A Framework for Distributed XML Data Management. In *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 1049–1058. Springer, 2006.

[6] B. Brumitt et al. EasyLiving: Technologies for intelligent environments. In *Proceedings of HUC 2000*, pages 12–29, 2000.

[7] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of CIDR'03*, 2003.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of SIGMOD'00*, pages 379–390, 2000.

[9] D. Estrin et al. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[10] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *Proceedings of SIGMOD'99*, pages 311–322, 1999.

[11] M. Franklin, A. Halevy, and D. Maier. From Databases to Dataspaces: a new Abstraction for Information Management. *SIGMOD Record*, 34(4):27–33, 2005.

[12] M. J. Franklin et al. Design Considerations for High Fan-In Systems: The HiFi Approach. In *Proceedings of CIDR'05*, 2005.

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

[14] R. Goldman and J. Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *Proceedings of SIGMOD'00*, pages 285–296, 2000.

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

[16] T. Imielinski and B. Nath. Wireless graffiti: data, data everywhere. In *Proceedings of VLDB'02*, pages 9–19, 2002.

[17] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. In *Pervasive*, pages 83–100, 2006.

[18] M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999.

[19] OSGi Alliance. `http://www.osgi.org/`.

[20] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query Optimization over Web Services. In *Proceedings of VLDB'06*, pages 355–366, 2006.

[21] UPnP Forum. `http://www.upnp.org/`.

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

[23] J. Widom et al. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[24] W. Xue and Q. Luo. Action-Oriented Query Processing for Pervasive Computing. In *Proceedings of CIDR'05*, 2005.

[25] Y. Yao and J. Gehrke. Query Processing in Sensor Networks. In *Proceedings of CIDR'03*, 2003.

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