World Scientific
www.worldscientific.com

C-SPARQL: A CONTINUOUS QUERY LANGUAGE
FOR RDF DATA STREAMS

DAVIDE FRANCESCO BARBIERI[*], DANIELE BRAGA[†], STEFANO CERI[‡],
EMANUELE DELLA VALLE[§] and MICHAEL GROSSNIKLAUS[¶]

*Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza L. Da Vinci, 32–20133 Milano, Italy*
[*]dbarbieri@elet.polimi.it
[†]braga@elet.polimi.it
[‡]ceri@elet.polimi.it
[§]dellavalle@elet.polimi.it
[¶]grossniklaus@elet.polimi.it

This article defines C-SPARQL, an extension of SPARQL whose distinguishing feature
is the support of continuous queries, i.e. queries registered over RDF data streams and
then continuously executed. Queries consider *windows*, i.e. the most recent triples of such
streams, observed while data is continuously flowing. Supporting streams in RDF format
guarantees interoperability and opens up important applications, in which reasoners can
deal with evolving knowledge over time.

C-SPARQL is presented by means of a full specification of the syntax, a formal
semantics, and a comprehensive set of examples, relative to urban computing applica-
tions, that systematically cover the SPARQL extensions. The expression of meaningful
queries over streaming data is strictly connected to the availability of aggregation prim-
itives, thus C-SPARQL also includes extensions in this respect.

Keywords: RDF; SPARQL; stream processing; continuous queries; stream reasoning.

1. Introduction

"How many cars are currently entering the city center?" "How many of them come
from the north-east district?" "Is the traffic in the streets of the north-east district
flowing smoothly?" Data Stream Management Systems (DSMS) [15] enable users to
issue such queries and are able to continuously answer them, by producing results
which are also varying with time. The revolutionary idea, which made it possible,
was to give up with one-time semantics, typical of database management systems,
and turn to continuous semantics.

In this paper, we address the challenges linked to a similar "revolution" in
the context of the Semantic Web and RDF repositories. As a matter of fact,
RDF repositories are scaling up in the classical time invariant domain of static

knowledge: SPARQL engines are now capable of querying integrated repositories, collecting data from multiple sources and supporting complex queries. However, the opportunity of combining static (or relatively slowly changing) RDF data with rapidly changing (or "streaming") information has been neglected or forgotten so far. Indeed, we deem that there is a potential interest for giving up with one-time semantics in RDF repositories as well, so as to explore the benefits provided by continuous semantics.

Therefore, we introduce *RDF streams* as the natural extension of the RDF data model to the new scenario continuous and Continuous SPARQL (or simply *C-SPARQL*) as the extension of SPARQL for querying RDF streams. Augmenting the knowledge bases with streaming facts is useful in all contexts in which reasoning on dynamic information is needed, like computing time-varying traffic conditions in a city or monitoring patient's parameters in medical applications. Standard reasoners cannot cope yet with dynamic knowledge, as they consider static RDF graphs to carry out reasoning tasks.

1.1. *Contributions and structure*

C-SPARQL is defined in terms of several orthogonal extensions of SPARQL, in such a way that a regular SPARQL query is also a C-SPARQL query.

- Similar to RDF graphs, every RDF stream is identified by using an IRI, but instead of being a static collection of RDF triples a stream is a timed sequence of RDF triples, continuously produced by a data source. Every RDF triple in a stream is annotated with a timestamp.
- The distinguishing feature of C-SPARQL is the support for continuous queries, i.e. queries that are registered over RDF streams and then executed continuously. C-SPARQL queries produce as output the same output types as SPARQL: boolean answers, selections of variable bindings, constructions of new RDF triples, or RDF descriptions of the involved resources. These outputs are continuously renewed with each query execution. In addition, C-SPARQL queries can be designated to produce new RDF streams.
- Extending SPARQL into C-SPARQL requires several additions for defining streams, their windows, their timestamps, and language constructs for aggregating stream information. In particular, query registration and designating the window structure are the key aspect of C-SPARQL.

The structure of the paper is as follows. Section 2 provides necessary background on DSMS and SPARQL. Section 3 introduces C-SPARQL by explaining in details the notions of RDF stream, window over an RDF stream, query registration, aggregates, and timestamp function. Section 4 describes the formal semantics of the language. Section 5 provides some running examples to illustrate the C-SPARQL syntax and informal semantics. Finally, Sec. 6 illustrates how we intend to progress with this research and Sec. 7 concludes the paper.

2. Background and Related Work

This section illustrates previous work on data streams and then on the SPARQL language.

2.1. *Data streams*

In many real-world applications data take the form of continuous streams instead of the form of finite data sets stored in a traditional repository. This is the case for monitoring of network traffic, for telecommunications management, for clickstreams, for manufacturing, for sensor networks, and for many other domains. In such applications, instead of classical "one-shot" queries, clients need to register continuously running queries, which return new results as new data arrive on the streams. A data stream is a sequence of items received continuously and in real-time, ordered either implicitly, by arrival time, or explicitly, by means of timestamps. Not only is it typically impossible to control the order in which items arrive, but, even more importantly, it is not feasible to locally store a stream in its entirety [16]. Due to all these peculiar characteristics, traditional database systems and data processing algorithms are not suitable for handling numerous and complex continuous queries over data streams. Ad-hoc data management systems have been studied and developed since the late nineties.

 One of the first proposed models for data streams was the Chronicle data model [23]. It introduced the concept of chronicles as append-only ordered sequences of tuples, together with a restricted view definition language and an algebra that operates over chronicles as well as over traditional relations. OpenCQ [27] and NiagaraCQ [10] addressed continuous queries for monitoring persistent data sets spread over wide-area networks. Another data stream management system is Aurora [6], which in turn evolved into the Borealis project [1], which addresses the distribution issues.

 In [4], Babu *et al.* tackle the problem of continuous queries over data streams addressing semantic issues as well as efficiency concerns. They specify a general and flexible architecture for query processing in the presence of data streams. This research evolved into the specification and development of a query language tailored for data streams, named CQL [2, 3]. Further optimizations are discussed in [28].

 A different perspective on the same issue brought Law *et al.* [25] to put particular emphasis on the problem of *mining* data streams [26]. Another DSMS is Stream Mill [5], which extensively considered and addressed data mining issues, specifically with respect to the problem of online data aggregation and to the distinguishing notion of blocking and non-blocking operators. Its query language (ESL) efficiently supports physical and logical windows (with optional slides and tumbles) on both built-in aggregates and user-defined aggregates. The constructs introduced in ESL extend the power and generality of Data Stream Management Systems.

 The problem of processing delays is one of the most critical issues and at the same time a strong quality requirement for many data stream applications, since

the value of query results decreases dramatically over time as the delays sum up. In [11], the authors address the problem of keeping delays below a desired threshold in situations of overload, which are common in data stream systems. The framework described in the paper is built on top of the Borealis platform.

As for the join over data streams, rewriting techniques are proposed in [17] for streaming aggregation queries, studying the conditions under which joins can be optimized and providing error bounds for results of the rewritten queries. The basis for the optimization is a theory in which constraints over data streams can be formulated and the result error bounds are specified as functions of the boundary effects incurred during query rewriting.

2.2. *SPARQL*

SPARQL has been developed under the patronage of the World Wide Web Consortium as the standard query language for RDF. Consequently, the most authoritative source about the syntax and semantics of the SPARQL language is the W3C recommendation [30].

Cyganiak [12] presents a relational model of SPARQL. The author uses relational algebra operators (join, left outer join, projection, selection, etc.) to model the SPARQL select clauses. A translation system between SPARQL and SQL is outlined. The system extensively resorts to the use of COALESCE and IS NULL in order to express some SPARQL features. Harris [22] presents an implementation of SPARQL queries over a relational database engine. The use of relational algebra operators is similar to that of [12].

In [8], a logical reconstruction of RDF family of languages is given. The authors prove the equivalence of their framework to W3C definition of RDF, getting complexity results and a model theoretic semantics.

In [20], Gutierrez *et al.* discuss the semantics and the computational complexity of a conjunctive query language for RDF with basic patterns, which is a formal and unambiguous basis for defining the semantics of SPARQL queries evaluation. In [29], Perez *et al.* consider simple RDF graphs (without special semantics for literals) and a simplified version of filters. These assumptions allow them (1) to provide a compositional semantics; (2) to prove that there is a normal form in which, under certain constraints over variable bindings, a wide range of queries can be expressed; (3) to fix some complexity bounds; and (4) to discuss optimization opportunities. Haase *et al.* [21] present a comparison of functionalities of pre-SPARQL query languages, many of which gave inspirations to the definition of SPARQL.

A previous attempt to extend SPARQL to support data streams has been presented by Bolles *et al.* [7]. Their paper represents an antecedent of our work as it introduces a syntax for the specification of logical and physical windows in SPARQL queries by means of local grammar extensions. However, their approach is different from the work presented in this paper in several key aspects. First, Bolles *et al.* simply introduce RDF streams as a new data type, and omit essential ingredients,

such as aggregate and timestamp functions. With these limitations, the resulting expressive power is not sufficient to express most practical queries. Second, the authors do not follow the established approach where windows are used to transform streaming data into non-streaming data in order to apply standard algebraic operations. Instead, Bolles *et al.* have chosen to change the standard SPARQL operators by making them timestamp-aware and, thereby, effectively introduce a new language semantics. Finally, their approach allows window clauses to appear within SPARQL group graph pattern expressions. On the one hand, this makes the query syntax more intricate, as window clauses can appear in multiple places. On the other hand, it complicates query evaluation. Since window operations are no longer required to be at the leaves of the query tree, they need to be interleaved with standard SPARQL operations, violating the separation of concerns between stream management and query evaluation.

In [35] the authors describe techniques of reasoning that can deal with streaming facts. Their work is complementary to ours, as they focus their work on advanced reasoning techniques that can scale up to high frequencies, rather than query processing and stream management.

3. C-SPARQL

This section presents C-SPARQL by first defining its syntax and then showing several examples of increasing complexity. Since the syntactic extensions are succinct and well-defined, we prefer to introduce all of them at once and then use a series of examples that progressively showcase the extensions.

We first characterize RDF streams as a new data type. Then we introduce the syntax for identifying streams and for defining windows over streams, followed by the definition of aggregation as an orthogonal extension. Finally, we discuss a function used for retrieving timestamp information. The syntax of each new feature is given in terms of additional productions to be added to the standard SPARQL grammar [31].

3.1. *RDF stream data type*

C-SPARQL adds *RDF streams* to the data types supported by SPARQL.[a] An RDF stream is defined as an ordered sequence of pairs, where each pair is constituted by an RDF triple and its timestamp τ:

$$\ldots$$
$$(\langle subj_i, pred_i, obj_i \rangle, \tau_i)$$
$$(\langle subj_{i+1}, pred_{i+1}, obj_{i+1} \rangle, \tau_{i+1})$$
$$\ldots$$

[a]Similarly, the stream type extends the relation type in CQL.

Timestamps can be considered as *annotations* of RDF triples; they are monotonically non-decreasing in the stream ($\tau_i \leq \tau_{i+1}$). They are not strictly increasing because timestamps are not required to be unique. Any (unbounded, though finite) number of consecutive triples can have the same timestamp, meaning that they "occur" at the same time, although sequenced in the stream according to some positional order.

3.2. *Windows*

The introduction of RDF streams as a new type of input data requires the ability to *identify* such data sources and to specify *selection* criteria over them.

For *identification*, we assume that each data stream is associated with a distinct IRI, that is a locator of the actual data source of the stream. More specifically, the IRI represents an IP address and a port for accessing streaming data.[b]

As for *selection*, given that streams are intrinsically infinite, we introduce the notion of windows upon streams, whose types and characteristics are inspired by those of the windows in continuous query languages such as CQL [3].

Identification and windowing are expressed in C-SPARQL by means of the FROM STREAM clause:

```
FromStrClause → 'FROM' ['NAMED'] 'STREAM' StreamIRI '[ RANGE' Window ']'
Window        → LogicalWindow | PhysicalWindow
LogicalWindow → Number TimeUnit WindowOverlap
TimeUnit      → 'ms' | 's' | 'm' | 'h' | 'd'
WindowOverlap → 'STEP' Number TimeUnit | 'TUMBLING'
PhysicalWindow → 'TRIPLES' Number
```

A window extracts from the stream the *last* data stream elements, which are considered by the query. Such extraction can be *physical* (a given number of triples) or *logical* (a variable number of triples which occur during a given time interval).

Logical windows are *sliding* [18] when they are progressively advanced of a given STEP (i.e. a time interval that is shorter than the window's time interval); they are *non-overlapping* (or TUMBLING) when they are advanced of exactly their time interval at each iteration. With tumbling windows every triple of the data stream is only considered into exactly one window, whereas with sliding windows some triples can be included into several windows.

The optional NAMED keyword works exactly as if applied to the standard SPARQL FROM clause for tracking the provenance of triples. It binds the IRI of a stream to a variable which is later accessible through the GRAPH clause.

---

[b]In a typical C-SPARQL application we expect that several streaming data sources can be available on the Internet, modeled as RDF data streams (possibly after data transcoding); RSS feeds or other publish-subscribe mechanisms allow to be seamlessly captured by the same abstraction.

3.3. *Query registration*

Continuous queries in C-SPARQL are queries over RDF data streams (i.e. queries including at least one FROM STREAM clause).

A registered C-SPARQL query produces continuous output in the form of variable bindings tables or graphs. Each C-SPARQL query is registered through the following statement.

Registration → 'REGISTER QUERY' *QueryName*
           ['COMPUTED EVERY' *Number TimeUnit*] 'AS' *Query*

The optional COMPUTED EVERY clause indicates the frequency at which the query *should* be computed. If no frequency is specified, the query is computed at a frequency that is automatically determined by the system.[c]

In addition, C-SPARQL allows the production of RDF streams, registered through the following statement.

Registration → 'REGISTER STREAM' *QueryName*
           ['COMPUTED EVERY' *Number TimeUnit*] 'AS' *Query*

We only register CONSTRUCT and DESCRIBE queries as STREAM, as they produce RDF triples that, once associated with a timestamp, yield RDF streams that can be managed in C-SPARQL. Every query execution produces from a minimum of one triple to a maximum of an entire RDF graph in output, depending on the query construction pattern. In the former case, a different timestamp is assigned to every triple, while in the latter case, the same timestamp is assigned to all the triples of the constructed graph. Still, the system-generated timestamps are in monotonic non-decreasing order.

3.4. *Aggregation in C-SPARQL*

The SPARQL specification lacks aggregation capabilities, but a continuous query language over streams without aggregates would not be useful in practice. Therefore, we have added aggregation to C-SPARQL.[d] The addition is orthogonal: the clauses supporting aggregates can syntactically and semantically be added to SPARQL, giving rise to a language extension which is fully autonomous and significant per se.

---

[c]Several Stream Management Systems are capable of self tuning the execution frequency of registered queries. This not only applies to queries with unspecified registration frequencies, but also whenever, due to peaks of workload, the execution frequency of all queries is reduced, so as to gracefully degrade the overall performances.
[d]Many SPARQL implementations already support aggregation, even in the absence of RDF streams.

We allow multiple independent aggregations within the same C-SPARQL query, thus pushing the aggregation capabilities beyond those of SQL.

Aggregation clauses have the following syntax.

AggregateClause → ('AGGREGATE { (' *var* ',' *Function* ',' *Group* ')' [*Filter*] '}')*
Function → 'COUNT' | 'SUM' | 'AVG' | 'MIN' | 'MAX'
Group → *var* | '{' *var* (',' *var*)* '}'

Every aggregation clause has the following three parts.

- The first is a new variable (i.e. a variable not occurring in the WHERE clause or in any other aggregation clause).
- The second is an aggregation function (one of: COUNT, MAX, MIN, SUM, AVG); COUNT may have no argument, while the other functions take one of the variables occurring in the WHERE clause as argument.
- The third is a set of one or more variables, occurring in the WHERE clause, which express the grouping criteria.

Every clause may also have an optional fourth part, a FILTER clause. The semantics of a query with aggregate functions consists in adding to the regular variable bindings computed by the WHERE clause some new bindings, one for each of the new variables introduced by the AGGREGATE clauses. The solution constructed in this way may be further filtered by a standard FILTER clause, which may refer to all the variables introduced in the WHERE and AGGREGATE clauses. The evaluations of aggregate functions are all independent from each other and take place after the computation of the bindings provided by the WHERE clause.[e]

3.5. *Timestamp function*

The timestamp of a stream element can be retrieved and bound to a variable using a timestamp function. The timestamp function has two arguments.

- The first is the name of a variable, introduced in the WHERE clause and bound by pattern matching with an RDF triple of that stream.
- The second (optional) is the URI of a stream, that can be obtained through SPARQL GRAPH clause.

The function returns the timestamp of the RDF stream element producing the binding. If the variable is not bound, the function is undefined, and any comparison involving its evaluation has a non-determined behavior. If the variable gets bound multiple times, the function returns the most recent timestamp value relative to the query evaluation time.

---

[e]We deliberately constrain C-SPARQL aggregates to use only the variables in the WHERE clause — and not other variables bound by other AGGREGATE clauses — in order to achieve their independence.

4. Formal Semantics of C-SPARQL

This section provides the formal semantics of C-SPARQL. In order to do this, we build on the work of Pérez *et al.* [29], and extend it with the formalization of aggregates, windows and the timestamp function. We address the reader to [29] for details and, for the sake of readability, only summarize the basic aspects of their formalization here.

The semantics of a C-SPARQL query is formalized via the concept of mapping. We denote as I, B, L, V respectively the domains of IRIs, blank nodes, literals, and variables which are all disjoint. We also define $T = (I \cup B \cup L)$. A *mapping* μ is a partial function $\mu : V \to T$ which gives the bindings for all the variables of a query. Evaluation occurs when a *graph pattern* (denoted as P) in the query is matched against an RDF dataset (D). P is a set of triple patterns $t = (s, p, o)$ such that $s, p, o \in (V \cup T)$. We then define $dom(\mu)$ as the subset of V where μ is defined (i.e., the domain of μ), and $deg(\mu)$ as the cardinality of $dom(\mu)$. We also use the notation $\mu(x)$ to refer to the bindings of variable x in μ. Two mappings μ' and μ'' are said to be *compatible* if $\forall x \in dom(\mu') \cap dom(\mu'')$, then $\mu'(x) = \mu''(x)$.

Let Ω_1 and Ω_2 be sets of mappings. Then the basic operators for the composition of mappings are

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible}\}$$
$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$$
$$\Omega_1 \backslash \Omega_2 = \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\}$$

The left outer-join is a derived operator

$$\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$$

The evaluation of a graph pattern P over a dataset D, is compactly denoted as $[[P]]_D$, and is defined recursively, as follows:

(1) $[[t]]_D = \{\mu \mid dom(\mu) = var(t) \wedge \mu(t) \in D\}$, where t is a triple pattern and $var(t)$ is the set of variables occurring in t
(2) $[[(P_1 \text{ AND } P_2)]]_D = [[P_1]]_D \bowtie [[P_2]]_D$
(3) $[[(P_1 \text{ OPT } P_2)]]_D = [[P_1]]_D \mathbin{⟕} [[P_2]]_D$
(4) $[[(P_1 \text{ UNION } P_2)]]_D = [[P_1]]_D \cup [[P_2]]_D$

4.1. *Aggregates*

We start by extending the binary operators (UNION, AND, OPT, and FILTER) with the new operator AGG (short for AGGREGATE). An *aggregation pattern* is denoted as $A(v, f, p, G)$, where v is the name of the new variable, f is the name of the aggregation function to be evaluated, p is the parameter of f, and G is the set of the grouping variables. We extend the evaluation of $[[P]]_D$ by adding a fifth rule to

the above definition to deal with aggregation patterns:

(5) $[[(P \text{ AGG } A)]]_D = [[P]]_D \bowtie [[A]]_D$, where P is a standard graph pattern and $A(v_a, f_a, p_a, G_a)$ is an aggregation pattern.

The evaluation of $[[A]]_D$ is defined by a mapping $\mu_a : V \to T$, where $dom(\mu_a) = v_a \cup G_a$; also, $deg(\mu_a) = deg(G_a) + deg(v_a) = deg(G_a) + 1$. This extension fully conforms to the notion of compatibility between mappings. Indeed, $v_a \notin dom(P)$ and, therefore, calling μ_p the mapping that evaluates $[[P]]_D$, μ_p and μ_a are compatible.

The result of the evaluation of μ produces a table of bindings, having one column for each variable $v \in dom(\mu)$. We can refer to a specific row in this table as $\mu_{(i)}$, and to a specific column as $\mu[v]$. The i-th binding of v is therefore $\mu_{(i)}[v]$.

The values to be bound to the variable v_a are computed as

$$\forall i \in [1, deg(\mu)], \quad \mu_{(i)}[v_a] = f_a(p_a, \mu[G_a])$$

where $f(p_a, \mu[G_a])$ is the evaluation of the function $f_a \in$ (SUM, COUNT, AVG, MAX, MIN) with parameters p_a over the groups of values in $\mu[G_a]$. The set of groups of values in $\mu[G_a]$ is made of all the distinct tuples $\mu_{(i)}[G_a]$, i.e., the subset of the mapping $\mu[G_a]$ without duplicate rows.

4.2. *Windows*

We define an *RDF stream* as

$$R = \{(\langle subj, \ pred, \ obj\rangle, \tau) \mid \langle subj, \ pred, \ obj\rangle \in ((I \cup B) \times I \times (I \cup B \cup L)), \tau \in \mathbb{T})\}$$

where $\mathbb{T}$ is the infinite set of timestamps. Note that triple patterns are enclosed in round brackets while triples are enclosed in angular brackets.

A *logical* window is defined as

$$\omega_l(R, t_i, t_f) = \{(\langle s, p, o\rangle, \tau) \in R \mid t_i < \tau \le t_f\}.$$

Let

$$c(R, t_i, t_f) = |\{(\langle s, \ p, \ o\rangle, \tau) \in R | t_i < \tau \le t_f\}|$$

be a function which counts the items in R which have timestamp in the range $(t_i, t_f]$, then a *physical* window is defined as

$$\omega_p(R, n) = \{(\langle s, \ p, \ o\rangle, \tau) \in \omega_l(R, t_i, t_f) | c(R, t_i, t_f) = n\}.$$

A window ω can be *sliding*, with *range* ρ and *step* σ. For logical windows, ρ and σ take the form of a time interval. Logical windows (a) contain the most recent triples in a time interval of length ρ; and (b) are evaluated with frequency $1/\sigma$. For physical windows, ρ and σ are integers. Physical windows (a) contain the last ρ triples; and (b) are evaluated whenever σ new triples arrive in the stream. A window is said to be as *tumbling* with range ρ if it is sliding with range ρ and step ρ.

4.3. *Timestamp function*

A variable v can occur multiple times in a graph pattern P. When P is matched against D, v gets as many bindings as are its occurrences in P. Some of these bindings may derive from static data, others from streaming data. Each of the bindings coming from the stream R is characterized by the timestamp of the triple that matches one of the triple patterns $t \in P$ such that $v \in dom(t)$. We denote the set of timestamps associated with a variable by a triple pattern t as $TS_{set}(v, t)$ and the set of all timestamps associated with the variable by a graph pattern P as

$$TS_{set}(v, P) = \{\tau | t \in P \land v \in dom(t) \land \tau \in TS_{set}(v, t)\}.$$

We can now define the timestamp function

$$ts(v, P) = \max(TS_{set}(v, P))$$

which returns the *highest* timestamp associated with v among all bindings of v in P. The timestamp function returns a value only if v has been matched at least once over a triple $\langle s, p, o \rangle \in R$, $\perp$ otherwise.

5. Examples of C-SPARQL

In this section, we progressively introduce examples of use of C-SPARQL in urban computing.

5.1. *Urban computing*

As cities grow and evolve, their state changes continuously under the influenced of human and natural factors. People moving in, out and within cities using a variety of means of transportation is one of the most easily observed human factors. Good examples for natural factors are weather changes, and their effects on pollution and street congestion.

Urban computing is a branch of Pervasive Computing[f] applied to everyday urban settings and lifestyle [24]. Streets, squares, restaurants, shops, train stations, subways, buses, cabs, and any semipublic space in our cities are examples of urban settings. Urban lifestyles are even broader and include people living, working, visiting and having fun in those settings. People constantly enter and leave urban settings with patterns of behavior that change between day and night.

So far, the application of Pervasive Computing to urban settings and lifestyles has been explored very little. Pervasive Computing is normally applied either to homogeneous large scale areas (e.g., forests, reefs, and glaciers) or, on the other hand, in small-scale ones (e.g., smart rooms and houses). In both of these application scenarios, researchers can deploy their own sensors or actuators and can rely on a

---

[f]Pervasive Computing is a human-computer interaction paradigm in which ICT is seamlessly integrated into everyday objects and activities.

well-defined environment. On the contrary, experiments in urban settings cannot assume a controlled environment. The core problem is no longer deploying sensing and actuation technologies, but rather building systems to interact with already deployed sensors and actuators. In many cases, urban computing even necessitates asking people to "act" as sensors or actuators.

Up until very recently, setting up urban computing experiments was almost impossible due to the lack of sensors and sensor data. Today, a large amount of the required information is made available on the Internet at almost no cost. Data about commercial activities and meeting places, about events scheduled in the city and their venues, about the position and speed of public transportation vehicles as well as about the availability of parking in specific areas is readily available in computerized form. Several interesting cases of people acting as sensors are already documented, for instance in Wikimapia and OpenStreetMap [19] people voluntarily provide geographic information, in [32] the presence and the movements of people is estimated using the density of communication and the number of cellular hand-overs in a mobile phone network.

Urban computing poses a set of different challenges, ranging from technological to social issues. Our experience in the field, lead of to believe that sustainable mobility is an exemplar case of urban computing. Mobility demands have been growing steadily for decades and will continue to do so in the future. For many years, the primary way of dealing with this increasing demand has been the increase of the roadway network capacity, by building new roads or adding new lanes to existing ones. However, financial and ecological considerations are posing increasingly severe constraints on this process. Hence, there is a need for additional intelligent approaches designed to meet the demand while, at the same time, more efficiently utilizing existing infrastructure and resources.

In [13], we give a comprehensive list of requirements for information processing in the case of urban computing. In particular, we highlight the need to cope with knowledge and data that change over the time. For instance, data relating to street names, landmarks, etc. changes very slowly, whereas the number of cars that go through a traffic detector during a five minutes interval changes very rapidly. We conclude that an urban computing system must at least be able to continuously reason upon the combination of slowly evolving knowledge, such as street topology, and data streams such as the current position of all means of public transport. In the following, we show examples of use of C-SPARQL that precisely combine those two types of information.

5.2. *A simple query with aggregation*

Given that aggregation is an orthogonal extension with respect to the other extensions regarding the management of streams, we start with a query having aggregates but no streams. The first query simply counts the number of sensors which are statically placed in the streets and returns the street and the number of sensors if more

than 5 sensors have been placed in the same street. The query is not continuous, hence it needs no registration.

```
PREFIX  c: <http://linkedurbandata.org/city#>

SELECT DISTINCT ?street ?total-sensor
WHERE { ?sensor c:placedIn ?street . }
AGGREGATE {( ?total-sensor, COUNT, {?street} )
            FILTER (?total-sensor > 5)}
```

The query is executed as follows. First, all pairs of bindings of sensors with their street are extracted from the repository, then the number of sensors located at each street is counted into the new variable `total-sensor` and each resulting pair is extended into a triple, then the triples which satisfy the filter predicate are selected, and finally distinct pairs of street and total sensors are projected.

5.3. *Querying streaming knowledge*

A classic example of stream management is the counting of the number of cars that enter the city center through the various tollgates around it. Assuming that each tollgate registers each car going through it, the next query counts how many cars each tollgate has been registering in the last 10 minutes. The sliding window is adjusted every minute.

```
REGISTER QUERY CarsEnteringCityCenterPerTollgate
         COMPUTED EVERY 1m AS

PREFIX  t: <http://linkedurbandata.org/traffic#>

SELECT DISTINCT ?tollgate ?passages
FROM STREAM <http://streams.org/milantollgates.trdf>
             [RANGE 10m STEP 1m]
WHERE { ?tollgate t:registers ?car . }
AGGREGATE {( ?passages, COUNT, {?tollgate} )}
```

The query is executed as follows. First, all pairs of bindings of tollgates with the car they register are extracted from the current window over the stream, then the number of cars registered by each tollgate is counted into the new variable `passages` and each resulting pair is extended into a triple that, finally, is projected as distinct pairs of tollgate and passages.

The window considers all triples that are in input for the last 10 minutes, and advances every minute. This means that new triples enter into the window and old tuples exit from the window every minute. Note that the result of the aggregation does not change during the slide interval, therefore also the query result does not

change during the slide interval; it changes instead at every slide change (i.e., every minute).

In this stream, as in all the streams that we will use in the examples of this article, the predicate of the triple (e.g. `t:register`) is fixed while the subject and object part of the triple (e.g., `?tollgate` and `?car`) are variable. Thus, a physical source for this stream will have items consisting of pairs of values. This arrangement is coherent with RDF repositories whose predicates are taken from a small vocabulary constituting a sort of schema, but C-SPARQL makes no assumption on variable bindings of its stream triples.

5.4. *Combining static and streaming knowledge*

For a more complex example, consider the case in which we want to count the number of car entering the city center from each district. We assume to have previously stored in an RDF repository: (a) into which districts a city is divided, (b) which streets belong to each district and (c) in which street each tollgate is placed. The next query counts how many cars entered the city from each district in the last 30 minutes; the sliding window is modified every five minute.

```
REGISTER QUERY CarsEnteringCityCenterPerDistrict
         COMPUTED EVERY 5m AS

PREFIX  c: <http://linkedurbandata.org/city#>
PREFIX  t: <http://linkedurbandata.org/traffic#>

SELECT DISTINCT ?district ?passages
FROM STREAM <http://streams.org/milantollgates.trdf>
            [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?district c:contains ?street .
        ?tollgate c:placedIn ?street . }
AGGREGATE {( ?passages, COUNT, {?district} )}
```

The query is executed as follows. As in the previous query, all pairs of bindings of tollgates with the car they register are extracted from the current window over the stream. A graph pattern is used to extract the pairs of bindings from the RDF repository that relate tollgates to the city district in which they are. Then, the number of cars registered by the tollgates in each district is counted and bound to the new variable `passages` and, finally, it is projected as distinct pairs of district and passages.

5.5. *Example of evaluation through semantics*

In this example we demonstrate the evaluation of the following C-SPARQL query through the use of the formal semantics introduced in Sec. 4. Note that the query is

a simplification of the previous one as it counts the passages of cars for each street and not per city district.

```
REGISTER QUERY CarsPassingThroughEachStreet
         COMPUTED EVERY 5m AS

PREFIX  c: <http://linkedurbandata.org/city#>
PREFIX  t: <http://linkedurbandata.org/traffic#>

SELECT ?car, ?tollgate, ?street, ?passages
FROM STREAM <http://streams.org/milantollgates.trdf>
            [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?tollgate c:placedIn ?street . }
AGGREGATE {( ?passages, COUNT, {?street} )}
```

In this query, we select data again both from a static knowledge dataset (S) containing the location of every tollgate and an RDF stream R representing the data retrieved by all the tollgates.

The query can be expressed as

$$P = t_1 \text{ AND } t_2 \text{ AGG } A_1$$

where

- $t_1 = $?tollgate registers ?car
- $t_2 = $?tollgate placedIn ?street
- $A_1 = $ A(?passages, count, ?car, ?street)
- $D = \omega_l(R, now - 10, now) \cup S$

According to the given formalization, P is evaluated as

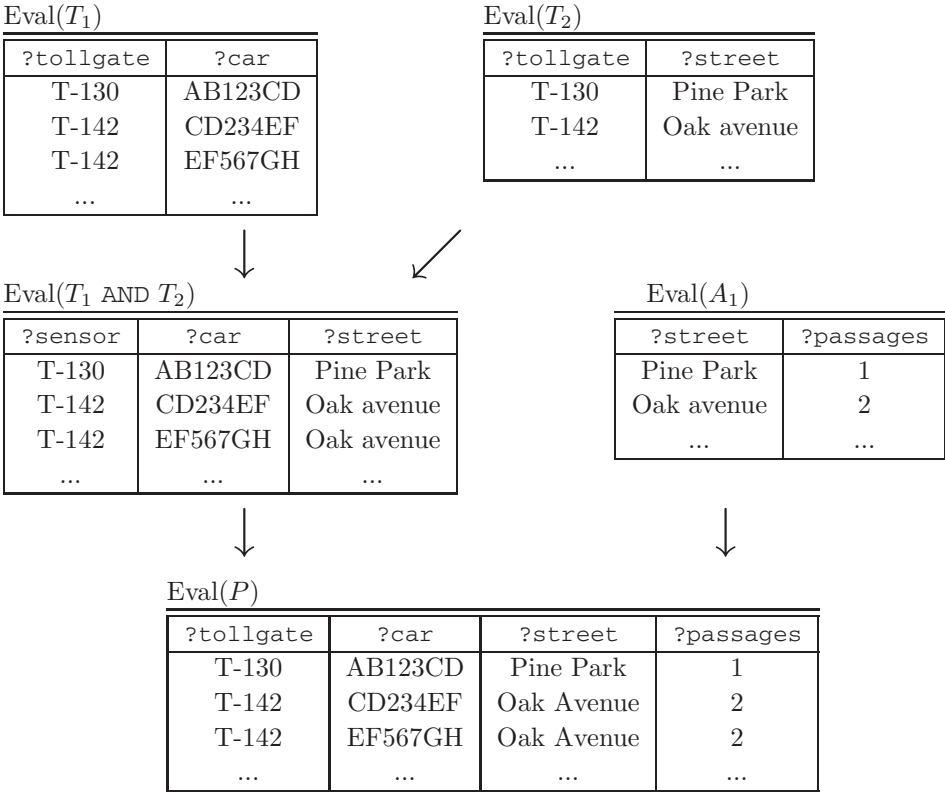$$[[P]]_D = [[t_1]]_D \bowtie [[t_2]]_D \bowtie [[A_1]]_D.$$

In particular, since t_1 and t_2 only take into account streaming data and static knowledge respectively, the evaluation of P can be rewritten as

$$[[P]]_D = [[t_1]]_R \bowtie [[t_2]]_S \bowtie [[A_1]]_D.$$

The evaluation steps are

(1) $T_1 = \text{Eval}(t_1)_D$
(2) $T_2 = \text{Eval}(t_2)_D$
(3) $T_3 = T_1 \text{ AND } T_2$
(4) $A_1 = \text{Eval}\left(\text{A(?total, count, ?car, ?street)}\right)_D$
(5) Join evaluation

Notice that the steps (1) and (2) can be evaluated in parallel. Based on a sample dataset, the tables below demonstrate the evaluation of $\text{Eval}(P)$ for each of the steps given above in terms of the corresponding intermediate result.

Eval(T_1)

?tollgate	?car
T-130	AB123CD
T-142	CD234EF
T-142	EF567GH
...	...

Eval(T_2)

?tollgate	?street
T-130	Pine Park
T-142	Oak avenue
...	...

Eval(T_1 AND T_2)

?sensor	?car	?street
T-130	AB123CD	Pine Park
T-142	CD234EF	Oak avenue
T-142	EF567GH	Oak avenue
...	...	...

Eval(A_1)

?street	?passages
Pine Park	1
Oak avenue	2
...	...

Eval(P)

?tollgate	?car	?street	?passages
T-130	AB123CD	Pine Park	1
T-142	CD234EF	Oak Avenue	2
T-142	EF567GH	Oak Avenue	2
...	...	...	...

5.6. *Queries with Multiple Aggregate Functions*

C-SPARQL supports multiple independent aggregations within one query. As an example, we combine the two aggregate functions used in the previous queries in one query, thus restricting car monitoring to those streets having at least five gates.

```
REGISTER QUERY CarsDetectedEnteringTheCityCenter
         COMPUTED EVERY 5m AS

PREFIX  c: <http://linkedurbandata.org/city#>
PREFIX  t: <http://linkedurbandata.org/traffic#>

SELECT DISTINCT ?district ?passages
FROM STREAM <http://streams.org/milantollgates.trdf>
             [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?district c:contains ?street .
        ?tollgate c:placedIn ?street . }
AGGREGATE {( ?passages, COUNT, {?district} )}
AGGREGATE {( ?total-sensor, COUNT, {?street} )
            FILTER (?total-sensor > 5)}
```

The two variables `total-sensor` and `passages` are both added to the variable binding table produced by the WHERE clause, and the filter condition eliminates from it those rows corresponding to streets with less than five sensors; then, the result of the query is extracted continuously in the same way as the previous example.

5.7. *Streaming the results of a query*

As continuous queries renew their output in each query execution, their output could in turn be periodically transferred to another system for further analysis (e.g., to plot the traffic as a function of time). In addition, C-SPARQL allows the construction of new RDF data streams, by supporting the possibility to register CONSTRUCT and DESCRIBE queries.

Consider the previous query in which the projected pairs of district and passages is used to construct a triple. The corresponding C-SPARQL query is

```
REGISTER STREAM CarsEnteringCityCenterPerDistrict
         COMPUTED EVERY 5m AS

PREFIX  c: <http://linkedurbandata.org/city#>
PREFIX  t: <http://linkedurbandata.org/traffic#>

CONSTRUCT {?district t:has-entering-cars ?passages}
FROM STREAM <http://streams.org/milantollgates.trdf>
             [RANGE 30m STEP 5m]
WHERE { ?tollgate t:registers ?car .
        ?district c:contains ?street .
        ?tollgate c:placedIn ?street . }
AGGREGATE {( ?passages, COUNT, {?district} )}
```

This query uses the same logical conditions as the previous one, but it constructs the output in the format of a stream of RDF triples. Every query execution may produce from a minimum of one triple to a maximum of an entire graph. In the former case, a different timestamp is assigned to every triple, while in the latter case, the same timestamp is assigned to all the triples of the graph. In both cases, timestamps are system-generated in monotonic order. Results of two evaluations of the previous query are presented in the table below.

Triple	Timestamp
`c:IT-MI-Baggio t:has-entering-cars "100"`	t_{400}
`c:IT-MI-Barona t:has-entering-cars "75"`	t_{400}
`c:IT-MI-Baggio t:has-entering-cars "130"`	t_{401}
`c:IT-MI-Barona t:has-entering-cars "95"`	t_{401}
`c:IT-MI-Fiera t:has-entering-cars "65"`	t_{401}

The first evaluation occurs at t_{400}. Suppose that only data from two districts (i.e., `c:IT-MI-Baggio` and `c:IT-MI-Barona`) are present in the window. Then, the evaluation generates two triples with the same timestamp (i.e., t_{400}).

The second evaluation occurs at t_{401}. Suppose that part of the data elaborated by the previous query are still in the window and that new data related to the district `uc:IT-MI-Fiera` entered in the window. Then, the evaluation produces three triples, all of them having the same new timestamp t_{401}.

5.8. *Combining multiple streams*

In addition to tollgates, assume that traffic control cameras also register cars at traffic lights and output a different stream. Based on the sum of cars seen by these cameras and passing toll gates, we consider a query returning all the streets which have been full for more than 80% of their capacity in the last 5 minutes.

```
REGISTER QUERY FullStreets AS

PREFIX   c: <http://linkedurbandata.org/city#>
PREFIX   t: <http://linkedurbandata.org/traffic#>

SELECT { ?street ?passages}
FROM STREAM <http://streams.org/milantollgates.trdf>
            [RANGE 5m TUMBLING]
FROM STREAM <http://streams.org/milancameras.trdf>
            [RANGE 5m TUMBLING]
WHERE {
  GRAPH <http://streams.org/milantollgates.trdf> {
    ?tollgate t:registers ?car .
    ?tollgate c:placedIn ?street   .
  } UNION
  GRAPH <http://streams.org/milancameras.trdf> {
    ?camera    t:registers ?car .
    ?camera    c:placedAt ?light .
    ?light     c:crossing ?street .
  } UNION {
    ?street    c:hasCapacity ?capacity . }
AGGREGATE {( ?passages, COUNT, {?street} )
          FILTER (?passages > (0.8 * ?capacity))}
```

The query is executed as follows. Pairs of bindings of tollgates with the cars are extracted from the first graph, using a window over the tollgate stream, and from the second graph, using a window over the control camera stream. Then, the capacity of each street is extracted from the RDF repository. At this point all the bindings are combined following the semantics of the UNION patter evaluation in SPARQL, and it becomes possible to count in the new variable passages the cars registered either by the tollgates or by the cameras in each street. Finally, the streets which

satisfy the filter predicate are selected and distinct pairs of street and passages are projected.

5.9. *Exploiting the timestamp function*

We now want to detect all cars turning from one street (via Golgi) into another one (via Celoria) by means of two cameras which are installed on the same traffic light. This is performed by the following query:

```
REGISTER STREAM AllCarsTurningFromGolgiIntoCeloria
          COMPUTED EVERY 1m AS

PREFIX  c: <http://linkedurbandata.org/city#>
PREFIX  t: <http://linkedurbandata.org/traffic#>

SELECT DISTINCT ?car1
FROM STREAM <http://streams.org/milancameras.trdf>
              [RANGE 5m STEP 1m]
WHERE { ?camera1 c:monitors c:MI-via-Celoria .
        ?camera2 c:monitors c:MI-via-Golgi.
        ?camera1 c: placedAt ?tr_light .
        ?camera2 c: placedAt ?tr_light .
        ?camera1 t: registers ? car1 .
        ?camera2 t: registers ? car2 .
        FILTER ( timestamp(?car1) > timestamp(?car2) AND
                 ?car1 = ?car2 )}
```

Note that we use the two variables so as to extract the two different timestamps, although they refer to the same physical car; this effect is obtained be the equality predicate in the FILTER clause. Then, the timestamp precedence checks that the car is seen by the first camera and then by the second camera. In this way, we match cars that are flowing in the right direction.

6. Future Work

The experiments conducted so far have been addressed to existing standard stream management systems, coupled with existing SPARQL engines, so as to be a proof of concept of the feasibility of the approach. Table 1 contains a comparative analysis of the stream related features of three prominent stream management systems, STREAM, Aurora/Borealis, and Stream Mill. For the future, we envision execution scenarios for C-SPARQL consisting of a collection of several interconnected nodes, each one supporting the execution of C-SPARQL queries. Nodes could have limitations/specializations, i.e. the ability of supporting pure SPARQL (with no data stream or aggregation) or specific stream-focused queries (with no capability of integration with a global RDF repository).

Table 1. Feature comparison of C-SPARQL and existing data stream management systems.

	Windows		
	Logical	Physical	Partitioning
C-SPARQL	✓	✓	
STREAM	✓*	✓	✓
Aurora/Borealis	✓	✓	
Stream Mill	✓	✓	✓
	Timestamps		
	Internal	External	Latent
C-SPARQL	✓		
STREAM	✓	✓	
Aurora/Borealis	✓	✓	
Stream Mill	✓	✓	✓
	Aggregation (1)		
	Functions	Custom Functions	Group-by
C-SPARQL	SQL-1†		✓
STREAM	SQL-1		✓
Aurora/Borealis	SQL-1	✓	✓
Stream Mill	SQL-1	✓	✓
	Aggregation (2)		
	Having	Multiple Aggregates	Blocking
C-SPARQL	✓	✓	Plug-in dependent
STREAM	✓		Yes
Aurora/Borealis	✓		Yes/No
Stream Mill	✓		Yes/No
	Architecture	Language	Semantics
C-SPARQL	Client/Server	SPARQL extension	Formal
STREAM	Client/Server	SQL extension	Operational
Aurora/Borealis	Network	XML dialect	Declarative
Stream Mill	Client/Server	SQL extension	Operational

* specified in the CQL language, but currently not supported by STREAM
† average, count, maximum, minimum, sum

Also, for what concerns query processing, several interesting optimization problems arise.

- The first issue concerns the development of analysis techniques for multiple queries, in order to recognize their common parts and reformulate the queries so as to compute common parts only once. This is a classical problem of continuous queries, known as multi-query optimization problem [33].
- Another issue concerns the development of incremental evaluation strategies that take into account the particular features of windows used in the query in order to compute the result at the next iteration as a function of the deltas that are produced due to incoming and outgoing triples in the window. This is also a classic problem of continuous queries [36].
- Another interesting optimization problem concerns the incremental materialization of knowledge that could be available to higher-order RDF-based reasoners (e.g., reasoners supporting RDF-S or OWL) when derived knowledge depends

upon streaming data. We can envision scenarios where knowledge at the next iteration could be computed as a function of the deltas that are produced due to incoming and outgoing triples in the window. This is an application if continuous queries to scenarios which support derived data, studied in [34].

- Another issue concerns the ability to adapt query computations under heavy query loads dynamically, e.g. by sampling input streams or by relaxing the window refreshment. Queries could be registered together with confidence limits so as to help the system's tuning. Sampling functions could be made available as part of the query language, thus enabling an explicit declaration within queries.
- Another issue concerns the distribution of query execution nodes close to data stream sources. The idea is that each source could provide input stream of arbitrary nature and then specific nodes could perform the transformation of such streams into summarized RDF triples and then transmit them to higher-level nodes for more complex computations. Nodes local to streams could support only stream-specific operations. Optimization could reuse classic results of distributed database systems [9].

7. Conclusions

The concept of C-SPARQL is simple, yet pervasive. While specialized Stream Database Management Systems provide solutions for on-the-fly analysis of data streams, they cannot combine data streams with knowledge bases, and cannot perform reasoning tasks. C-SPARQL bridges data streams to reasoning and enables stream reasoning, a new, unexplored, high impact research area.

C-SPARQL is our contribution to LarKC, an FP7 project sponsored by the EU for enabling massive and scalable reasoning computations [14]. The study of C-SPARQL execution environment is suggestive of several important research directions, sketched in the previous section, which may reuse and adapt classic query optimization results. In the course of the larKC project, we will choose the most promising and interesting ones and concentrate on them.

Acknowledgments

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing and S. Zdonik, The design of the borealis stream processing engine, in *Proc. Intl. Conf. on Innovative Data Systems Research* (*CIDR 2005*), 2005.

[2] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein and J. Widom, STREAM: The stanford stream data manager (Demonstration Description), in *Proc. ACM Intl. Conf. on Management of Data* (*SIGMOD 2003*), 2003, p. 665.

[3] A. Arasu, S. Babu and J. Widom, The CQL continuous query language: Semantic foundations and query execution, *The VLDB Journal* **15**(2) (2006) 121–142.

[4] S. Babu and J. Widom, Continuous queries over data streams, *SIGMOD Rec.* **30**(3) (2001) 109–120.

[5] Y. Bai, H. Thakkar, H. Wang, C. Luo and C. Zaniolo, A data stream language and system designed for power and extensibility, in *Proc. Intl. Conf. on Information and Knowledge Management (CIKM 2006)* 2006, pp. 337–346.

[6] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts and S. Zdonik, Retrospective on Aurora, *The VLDB Journal* **13**(4) (2004) 370–383.

[7] A. Bolles, M. Grawunder and J. Jacobi, Streaming SPARQL — Extending SPARQL to process data streams, in *Proc. Europ. Semantic Web Conf.* (*ESWC 2008*), 2008, pp. 448–462.

[8] J. D. Bruijn, E. Franconi and S. Tessaris, Logical reconstruction of normative RDF, in *Proc. Workshop on OWL: Experiences and Directions* (*OWLED 2005*), 2005.

[9] S. Ceri and G. Pelagatti, *Distributed Databases Principles and Systems* (McGraw-Hill, New York, 1984).

[10] J. Chen, D. J. DeWitt, F. Tian and Y. Wang, NiagaraCQ: A scalable continuous query system for internet databases, in W. Chen, J. F. Naughton and P. A. Bernstein, editors, *Proc. ACM Intl. Conf. on Management of Data* (*SIGMOD 2000*), 2000, pp. 379–390.

[11] Y. Cheng Tu, S. Liu, S. Prabhakar and B. Yao, Load shedding in stream databases: A control-based approach, in *Proc. Intl. Conf. on Very Large Data Bases (VLDB 2006)*, 2006, pp. 787–798.

[12] R. Cyganiak, A relational algebra for SPARQL, Technical Report, HP-Labs.

[13] E. Della Valle, I. Celino, D. DellAglio, K. Kim, Z. Huang, V. Tresp, W. Hauptmann, Y. Huang and R. Grothmann, *Urban Computing: A Challenging Problem for Semantic Technologies* **12** (2008).

[14] D. Fensel, F. van Harmelen, B. Andersson, P. Brennan, H. Cunningham, E. D. Valle, F. Fischer, Z. Huang, A. Kiryakov, T. K. Il Lee, L. School, V. Tresp, S. Wesner, M. Witbrock and N. Zhong, Towards LarKC: A platform for web-scale reasoning, in *Proc. IEEE Intl. Conf. on Semantic Computing (ICSC 2008)*, 2008.

[15] M. Garofalakis, J. Gehrke and R. Rastogi, *Data Stream Management: Processing High-Speed Data Streams* (*Data-Centric Systems and Applications*), (Springer-Verlag, New York, 2007).

[16] L. Golab, D. DeHaan, E. D. Demaine, A. López-Ortiz and J. I. Munro, Identifying frequent items in sliding windows over on-line packet streams, in *Proc. Intl. Conf. on Internet Measurement* (*IMC 2003*), 2003, pp. 173–178.

[17] L. Golab, T. Johnson, N. Koudas, D. Srivastava and D. Toman, Optimizing away joins on data streams, in *Proc. Intl. Workshop on Scalable Stream Processing System* (*SSPS 2008*), 2008, pp. 48–57.

[18] L. Golab and M. T. Özsu, Processing sliding window multi-joins in continuous queries over data streams, in *Proc. Intl. Conf. on Very Large Data Bases* (*VLDB 2006*), 2003, pp. 500–511.

[19] M. Goodchild, Citizens as sensors: The world of volunteered geography, *GeoJournal*, **69**(4) (2007) 211–221.

[20] C. Gutierrez, C. Hurtado and A. O. Mendelzon, Foundations of semantic web databases, in *Proc. ACM Symp. on Principles of Database Systems* (*PODS 2004*), 2004, pp. 95–106.

[21] P. Haase, J. Broekstra, A. Eberhart and R. Volz, A comparison of RDF query languages, in *Proc. Intl. Semantic Web Conf.* (*ISWC 2004*), 2004, pp. 502–517.

[22] S. Harris, SPARQL Query processing with conventional relational database systems, in *Proc. Intl. Workshop on Scalable Semantic Web Knowledge Base Systems* (*SSWS 2005*), 2005, pp. 235–244.

[23] H. V. Jagadish, I. S. Mumick and A. Silberschatz, View maintenance issues for the chronicle data model, in *Proc. ACM Symp. on Principles of Database Systems* (*PODS 1995*), 1995, pp. 113–124.

[24] T. Kindberg, M. Chalmers and E. Paulos, Guest editors' introduction: Urban computing, *IEEE Pervasive Computing* **6**(3) (2007) 18–20.

[25] Y.-N. Law, H. Wang and C. Zaniolo, Query languages and data models for database sequences and data streams, in *Proc. Intl. Conf. on Very Large Data Bases* (*VLDB 2004*), 2004, pp. 492–503.

[26] Y.-N. Law and C. Zaniolo, An adaptive nearest neighbor classification algorithm for data streams, in *Proc. Europ. Conf. on Principles and Practice of Knowledge Discovery in Databases* (*PKDD 2005*), 2005, pp. 108–120.

[27] L. Liu, C. Pu and W. Tang, Continual queries for internet scale event-driven information delivery, *IEEE Trans. Knowl. Data Eng.* **11**(4) (1999) 610–628.

[28] K. Munagala, U. Srivastava and J. Widom, Optimization of continuous queries with shared expensive filters, in *Proc. ACM Intl. Symp. on Principles of Database Systems* (*PODS 2007*), 2007, pp. 215–224.

[29] J. Pérez, M. Arenas and C. Gutierrez, Semantics and complexity of SPARQL, in *Proc. Intl. Semantic Web Conf.* (*ISWC 2006*), 2006, pp. 30–43.

[30] E. Prud'hommeaux and A. Seaborne, SPARQL Query language for RDF, http://www.w3.org/TR/rdf-sparql-query/.

[31] E. Prud'hommeaux and A. Seaborne, SPARQL Query language for RDF grammar, http://www.w3.org/TR/rdf-sparql-query/#sparqlGrammar.

[32] J. Reades, F. Calabrese, A. Sevtsuk and C. Ratti, Cellular census: Explorations in urban data collection, *IEEE Pervasive Computing* **6**(3) (2007) 30–38.

[33] P. Roy, S. Seshadri, S. Sudarshan and S. Bhobe, Efficient and extensible algorithms for multi query optimization, *SIGMOD Rec.* **29**(2) (2000) 249–260.

[34] R. Volz, S. Staab and B. Motik, Incremental maintenance of materialized ontologies, in *Intl. Conf. on Ontologies and Databases* (*ODBASE 2003*), 2003.

[35] O. Walavalkar, A. Joshi, T. Finin and Y. Yesha, Streaming knowledge bases, in *Proc. Intl. Workshop on Scalable Semantic Web Knowledge Base Systems* (*SSWS 2008*), 2008.

[36] J. Yang and J. Widom, Incremental computation and maintenance of temporal aggregates, *The VLDB Journal* **12**(3) (2003) 262–283.