Real-Time Urban Monitoring in Dublin Using Semantic and Stream Technologies

Simone Tallevi-Diotallevi[1,2], Spyros Kotoulas[1], Luca Foschini[2], Freddy Lécué[1], and Antonio Corradi[2]

[1] Smarter Cities Technology Centre, IBM Research, Ireland
[2] Dip. Informatica Scienza e Ingegneria, DISI - Università di Bologna, Italy

Abstract. Several sources of information, from people, systems, things, are already available in most modern cities. Processing these continuous flows of information and capturing insight poses unique technical challenges that span from response time constraints to data heterogeneity, in terms of format and throughput. To tackle these problems, we focus on a novel prototype to ease real-time monitoring and decision-making processes for the City of Dublin with three main original technical aspects: (i) an extension to SPARQL to support efficient querying of heterogeneous streams; (ii) a query execution framework and runtime environment based on IBM InfoSphere Streams, a high-performance, industrial strength, stream processing engine; (iii) a hybrid RDFS reasoner, optimized for our stream processing execution framework. Our approach has been validated with real data collected on the field, as shown in our Dublin City video demonstration. Results indicate that real-time processing of city information streams based on semantic technologies is indeed not only possible, but also efficient, scalable and low-latency.

1 Introduction

Smarter Cities make available several resources to be managed and harnessed safely, sustainably, cost-effectively, and efficiently to achieve positive and measurable economic and societal advantages. Information gathered from people, systems, and things is one of the most valuable resource available to city stakeholders, but its enormous dimensions makes difficult its integration and processing, especially in a real-time and scalable manner. To tackle these challenges, stream reasoning merges and exploits synergies and common strategies of different disciplines, such as Machine Learning, Semantic Web, Databases, Data Mining, and Distributed Systems communities [1]. Stream reasoning aims to answer the several socio-technical questions about event monitoring and management by providing abstractions, foundations, methods, and tools required to deal with data streams in a scalable way [2].

Public Administration and Government are embracing Open Data, an important effort to expose any information with the final goal of increasing transparency and improving accountability of public services [3]. Along that direction, some cities including Dublin are publishing open data about several city dimensions such as transportation, environment, energy, and planning. In addition, Web sources are providing us with an abundance of information ranging from public bike availability [4] to weather information [5]. Non-public data can also be made available, such as pedestrian counts and information about the current location and state of public transportation resources. Those data are typically provided as machine-readable, though not machine-understandable, formats (e.g., CSV, xls, JSON). At the same time, ongoing seminal initiatives are trying to increase the utility and usability of this immense data repository by linking data through semi-structured machine-understandable formats. A good example of this trend is RDF-augmented information about points of interest (through LinkedGeoData [6] and DBPedia [7]). However, it is still unclear how to integrate raw and RDF-based data efficiently, as well as static data and data streams.

The end goal is to capture the spatial and temporal pulse of cities and make the city run better, faster, and cheaper. This goal requires tackling several challenges: (i) integration of heterogeneous (in terms of formats, throughput, etc.) open and external data and knowledge from different domains; (ii) high performance required to process a large volume of real-time data; and (iii) reasoning mechanisms to exploit the semantics of the data and simplify the search space.

We address all above issues to embrace the full potential of real-time reasoning for Smarter Cities, by proposing a solution that exhibits several novel characteristics. First, we define and include new operators and extensions to existing "streaming" SPARQL languages [8,9,10] to enable simultaneous processing of RDF data streams and raw data streams. Second, our system interfaces with IBM InfoSphere Streams [11] to enable continuous and fast analysis of massive volumes of real-time and heterogeneous (format, throughput, etc.) data. Third, we develop and benchmark a native RDFS reasoner, optimized for our stream processing execution framework, that allows us to automatically extract "hidden" information from Linked Data sources and exploits the formal semantics of RDF. Last, we validate our approach and illustrate the business value of our work in a real urban information-based scenario. We show real-time evaluation of Key Performance Indicators (KPIs) to capture the evolution of quality of life. In addition, at real-time, we can detect and rank abnormal situations in the Dublin City by reasoning over linked data and raw data, integrated on-the-fly. In both cases, our system exhibits high-throughput and low latency.

2 Related Work

Enabling city information as a utility requires sustainable technologies for composing and exposing real-time, big, and noisy data in an expressive and scalable way. Capturing the pulse of cities in real-time, as the high-level challenge we address in this work, requires the fusion and interpretation of large and open data. The application domain is vast, and there have been several interesting approaches, ranging from traffic diagnosis to water monitoring, but, in the interest of space, in this paper, we will only cover the ones that pertain to fusion and interpretation of data streams.

Although general approaches, using existing data mining [12], stream processing [11], machine learning approaches [2], have been presented for detecting, visualizing and analyzing patterns in data stream, they all fail in merging and analyzing large and heterogeneous sources of data in a real-time time context such as the motivating scenarios in Section 3. A set of systems extending basic database models for streams, such as Aurora [13], OpenCQ [14], Stream Mill [12], TelegraphCQ [15], focus on relational models, inhibiting their use in heterogeneous environments and do not focus on real-time stream integration.

From a Linked Data perspective, Streaming SPARQL [16], Time annotated SPARQL [17], EP-SPARQL [8], CQELS [9] and C-SPARQL [10] and other [18] extend SPARQL to manage RDF-based data streams. However integration of CSV-like data is not supported in such methods, requiring an additional, inefficient conversion step. In addition, these approaches do not support customized joins of stream and static data, making data fusion complex and inefficient. Finally, they all rely solely on an RDF representation, which, although very expressive, does not ensure fast analysis of massive volumes of real-time data. Our approach marries Linked Data and tuple-based processing, in order to provide the flexibility and expressive semantics of the first, without sacrificing much of the performance of the second.

3 Use-Case: Urban Monitoring

Cities increasingly rely on information management systems for decision-making. Decision-making can be roughly split into four main categories [19]: *strategic*, operating on long-term and at high aggregation level to evaluate and influence sustainability and growth, such as planning new development areas; *tactical*, targeting goals with a time horizon from days to months, such as preparing for snow in the winter; *operational*, that address events in a time-frame from minutes to hours, such as monitoring occupation of bike-sharing stations around the city; and *real-time*, time-critical operations on a frame from seconds to minutes, such as monitoring traffic to operate traffic lights.

We consider *operational* and *real-time* decisions by focusing on two real use case scenarios for Dublin City: At an operational level, calculating quality-of-life indicators, based on several dimensions such as pollution (air, noise), transportation (bus, bike-sharing). At a real-time level, improving public safety by providing a lightweight method to select (and cycle through) the most relevant Closed-Circuit television (CCTV) cameras to monitor at any given time. In the following, we give more details about the two considered scenarios that inspired our research and prototype. Although these scenarios and the corresponding technology we have developed pave the way for several business solutions for cities, this paper addresses solely the technological aspects.

Operational. At the operational level, dashboards are an emerging paradigm to communicate information efficiently at a high level of aggregation in business and in government [20]. We have created a scenario where a city manager would get an easy-to-read visual and geo-localized indication of some quality-of-life KPIs. In particular, we considered the following KPIs: environment, transportation, and an aggregate KPI, encompassing the other two. Elements in the *environment* are available from the city sensor infrastructure and we define our environment KPIs as weighted sums of sensor readings for pollution, noise, precipitation, and difference in temperature from a nominal value of 20 degrees Celsius. The performance of the city with regard to *transportation* can be measured by the performance of rapid transit, traffic situation, availability of alternative transportation means (such as bike-sharing schemes), and the number of people that are estimated to use such systems at any given time. In many situations, it is possible and desirable to aggregate KPIs so as to provide a higher-level view. For example, by combining the environment and transportation KPIs, one could draw a more general picture of the functioning of the city.

Real-Time. At the real-time level, the proliferation of CCTV systems for public safety and their decreasing cost have increased the number of cameras deployed and operated on behalf of government authorities: in Dublin alone, there are approximately 700 CCTV cameras for public safety, making it very difficult to process all generated video feeds in real-time. One option is using machine vision on the video streams, and there are indeed effective solutions for this [21]. Nevertheless, they do not cover the situation where the cause of the change is not immediately visible. In addition, unless image-detection algorithms can be locally executed, this solution incurs scalability problems due to the high bandwidth required to transmit video to the video collection endpoint. An additional issue is that the computational cost for the execution of effective machine vision methods is significant, requiring powerful compute clusters.

Hence, there is a need for lightweight methods to select the best cameras to monitor. We are considering the possibility to select cameras based on information about their surroundings and on changes in their environment, as detected by sensors close to them. In this sense, a "sensor" has a very wide meaning ranging from physical sensors capturing noise, to Web sensors producing streams about happenings in a city. Our decision-making process generally is as follows: (i) take into account a number of stream measurements, such as percentage of vehicles entering a region with a traffic congestion, ambient noise beyond a given threshold, etc., and assign a score to each of them, weighted by the distance from the cameras; (ii) assign a weight for the presence of amenities in the area, such as schools and hospitals; (iii) detect changes across three different time spans, called *windows*. We use a short-window of a few seconds to measure recent changes, a medium-window of tens of seconds to measure the persistence of the state evaluated by the short window, and a large-window, ending in the past, to account for regular variations, such as daily rush hours.

The information to support the above neither comes from a single source nor is in a single format. To make matters worse, some of the required information is "hidden" and needs to be made explicit (e.g. even though we may know that a facility is a hospital, it might not be explicit that it is also sensitive infrastructure). In Section 5, we give a more thorough description of the streams, datasets, formats to support our use case.

We conclude this section by anticipating the set of features we need in order to support the two scenarios introduced above. First of all, both scenarios are dealing with data from several sources, likely to be part of data source hierarchies, and require to *aggregate* and *infer* new information; at the same time, they both imply some sort of *scoring* and *ranking* based on aggregated data and KPIs. In addition, we would need to maintain multiple *windows* and *window types*. For example, with short vs long range, tumbling vs sliding windows (i.e. non-overlapping windows with no gap in-between vs overlapping windows) and sampling windows (windows with gaps in between). In the following Section, we are presenting a system with these features.

4 Realtime Reasoning and Continuous Query Answering

This section presents the three main technical contributions of our system towards efficient support of the scenarios described in Section 3: (i) *efficient processing of heterogeneous streams* with the possibility to directly aggregate input coming from RDF files and CSV streams; (ii) *real-time computation* with low latency and high throughput; (iii) *hybrid stream reasoning* by jointly: reducing the search space through backward reasoning (at query level), and joining results with materialized *relevant* static knowledge. The latter is used to prune the search space on the streams in order to increase the overall system performance.

Section 4.1 introduces our novel Dublin Extensions (DubExtensions) to the SPARQL grammar that enable the handling of heterogenous streams. Section 4.2 details the mapping from the SPARQL and window-based algebra to native operators in Infosphere Streams. Section 4.3 presents an optimized stream reasoning method.

4.1 DubExtensions

C-SPARQL and CQELS are SPARQL extensions to deal with RDF data streams [8,22]. In SPARQL, the main access method to data is through so-called Basic Graph Patterns (BGPs). BGPs consist of triple patterns and define the joins that need to be performed in a query. For example, a BGP may consist of the triple patterns <*?x :type :Bus.*> and <*?x :latitude ?y.*>. The first triple pattern will retrieve all the *?x*'es that are buses while the second will retrieve everything that has a latitude and the corresponding latitude value. The BGP for these two triple patterns will be the conjunction of the two (namely the latitudes of all busses) and it can be evaluated by performing an inner join on *?x*.

In this paper, we are proposing a set of extensions to SPARQL, called *DubExtensions*, incorporating elements of C-SPARQL and CQELS and support for direct processing of heterogeneous streams. More precisely, DubExtensions supports three BGP types: **Standard BGPs** that deal with static RDF data and represent the standard SPARQL BGPs; **Stream BGPs** that deal with RDF stream data [8]; and **CSV BGPs** that deal with mapping CSV-based stream data in non-RDF format to RDF format via pre-defined predicates. Although in this paper we limit our discussion to the CSV format, and refer to these BGPs as *CSV BGPs*, our architecture allows extensions to deal with any input that can be expressed in sets of tuples. In addition, as we will describe, CSV BGPs are restricted to triple patterns with particular predicates.

$$\begin{aligned}
\textbf{DatasetClause} &\rightarrow \textbf{FromClauseStream} \mid \textbf{FromClauseCSV} \mid \textbf{FromClauseOntology}\\
\textbf{FromClauseStream} &\rightarrow \text{'FROM ['NAMED'] 'STREAM' StreamIRI Number ['RANGE' Window] 'AS' Label}\\
\textbf{FromClauseCSV} &\rightarrow \text{'FROM CSV' StreamIRI Number ['RANGE' Window] 'AS' Label}\\
\textbf{FromClauseOntology} &\rightarrow \text{'FROM ONTOLOGY' StreamIRI}
\end{aligned} \tag{1}$$

First of all, to enable our new CSV BGP, we define new clauses to integrate CSV resources as a new type of input. The set of equations (1) shows our syntax extensions, allowing us not only to deal with stream heterogeneity (by supporting multiple input types), but also to improve system performance. Toward these goals, our DubExtensions introduce two main additional parameters in the FROM clause in order to interpret the input. In (1), *Number* specifies the CSV field of the timestamp bound to the specific

CSV stream, while *Label* specifies how to bind an input, and its relative window configuration, with the specific graph pattern inference query. In other words, *Label* allows us to refer to an input stream specification within the scope of the entire query. The label is not necessarily bound to a single input stream, but it can be shared between different inputs, but in that case, they must provide the same window configuration.

GraphPatternNotTriples → **GroupOrUnionGraphPattern** | **OptionalGraphPattern** | **MinusGraphPattern** | **GraphGraphPattern** | **StreamGraphPattern** | **CsvGraphPattern** | **ServiceGraph-Pattern** | **Filter** | **Bind**
StreamGraphPattern →'STREAM' Label '{'TriplesTemplate'}'
CSVGraphPattern →'CSV' Label '{'RestrictedTriplesTemplate'}'
(2)

Once the input is defined, we use it to configure new graph patterns (2); in particular, we exploit the *StreamGraphPattern* definition, introduced in CQELS [22], by extending it with the concept of label and we define the *CSVGraphPattern*, identified by the CSV keyword. The triple patterns specified within *CSVGraphPatterns* are restricted: We only allow predicates to bind specific columns of a CSV stream (e.g. *ibm:csvCol_2*, to bind the variable to the second column of the CSV). We omit the full definition of **RestrictedTriplesTemplate** in the interest of space. For all types of input streams, we favor explicit binding of BGPs to streams because it is beneficial for performance, since we are limiting the scope/number of joins.

Figure 1 shows an example of a DubExtensions-based query expressed by using our CSVGraphPattern. Every triple pattern is used to define a single variable of the query: the first element of the triple is the variable that will bound to the value of the specific field of the CSV stream, the second element is a special predicate ("*ibm:csvCol*") plus a number that is used to specify the field of the CSV record. The last element is the URI reference of the specific CSV input stream. For example, in line 12, in Figure 1 the variable "*?stationid*" will get its bindings for field zero ("*ibm:csvCol_0*") of the CSV stream identified by the URI $http://.../csv$.

```
1   SELECT ?station ?stationid ?stationlat ?stationlong ?address
2                       (AVG( ?bikecount / ?numBike ) AS ?bike)
3    FROM <http://.../rfid>
4    FROM CSV <http://.../csv> 1 [RANGE 20m STEP 20m] AS 'bikestream'
5    WHERE{
6      { ?station dpPedia:agencyStationCode ?stationid.
7        ?station dpPedia:maxNumOfBike ?numBike.
8        ?station wsg84:long ?stationlong.
9        ?station wsg84:lat ?stationlat.
10       ?station address:streetAddress ?address.}
11     CSV 'bikestream' {
12      ?stationid ibm:csvCol_0 <http://.../csv>.
13      ?bikecount ibm:csvCol_2 <http://.../csv>.}
14   } GROUP BY ?station ?stationid ?stationlat ?stationlong ?address
```

Fig. 1. Graph pattern extensions

Let us note that DubExtensions enable direct merging of static RDF and a dynamic CSV data stream, thus simplifying the definition of new queries to evaluate quality-of-life KPIs: the query shown in Figure 1 evaluates the average usage of bikes for each bike-sharing station every 20 minutes, by selecting the static information about stations ("*?stationid*") from a standard BGP, and by counting the number of bikes available ("*?bikecount*") from our CSV BGP.

Similarly, DubExtensions support similar windowing semantics as C-SPARQL. For every stream, we can define a range r and a step s. s refers to how often we will start a new window while r refers to the time period a window is kept. With these definitions, we can support various window types such as sliding windows ($s < r$) and tumbling windows ($s = r$) or only take fragments of the stream ($s > r$).

4.2 DubExtensions on InfoSphere Streams

In this Section, we outline some key points in translating SPARQL with DubExtensions to IBM InfoSphere Streams. InfoSphere Streams is a clustered stream processing platform that allows to easily define a programs as a graph of processing elements, called *operators*, expressed in the Stream Processing Language (SPL) [23]. SPL abstracts from operator implementation details but still allows users to control the most performance-critical stream processing aspects: drawing the graph topology, the stream connections (connections between operators) and tuning of data representation and operators.

We have used SPL to translate DubExtensions queries into high-performance streaming applications, consisting of the three main parts shown in Figure 2:

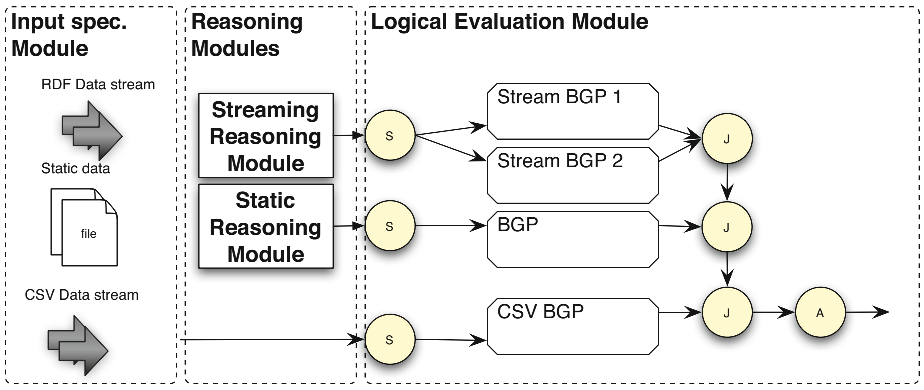
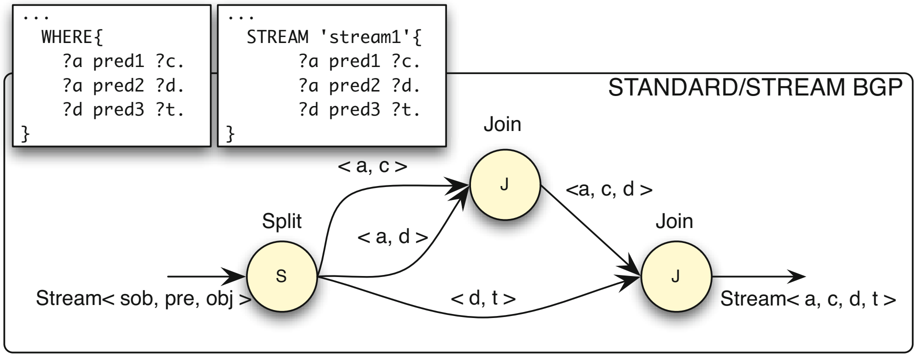


Fig. 2. Example application

Input Specification Module (ISMod). Consumes all inputs and feeds the other modules by labeling inputs according to the specification on the *Labels* in the FROM clause. In addition, for Stream BGPs and Standard BGPs, *ISMod* also performs filtering and projection operations to extract relevant data, discarding as soon as possible all data that is not necessary for the query. Since with RDF data it is very common to perform self-joins, the S operators allows splitting and/or replicating data according to specified query input patterns (see Figure 3). For CSV input, it also executes limited pre-processing operations such as extracting the tuples from the sub-set of specified columns and renaming them according to indexes and labels provided in the query.



```
...
  WHERE{
    ?a pred1 ?c.
    ?a pred2 ?d.
    ?d pred3 ?t.
}
```

```
...
  STREAM 'stream1'{
       ?a pred1 ?c.
       ?a pred2 ?d.
       ?d pred3 ?t.
}
```

Fig. 3. RDF BGPs

Reasoning Modules (InfMod). Performs reasoning over static and dynamic inputs. Inferencing over static data is performed using standard methods from the literature, although, as we will show later, we perform reasoning only on data that could possibly contribute to query results (see the WHERE clause in Figure 3). For dynamic data (STREAM clause), we apply an optimized reasoning method specifically designed for streams, detailed in Section 4.3.

Logical Evaluation Module (LEMod.) Composes the set of SPL operators that execute the relational processing and aggregation and manages query windowing. SPL relational operators largely support the SPARQL ones, so the mapping is relatively straightforward; similarly, SPL supports a super-set of the aggregation operators available in SPARQL 1.1. A full mapping between SPARQL 1.1 and SPL operators, however, is beyond the scope of this paper and difficult to obtain, especially for involved cases such as SPARQL property paths. Windowing clauses are mapped to SPL using a combination of time, count and punctuation-based constructs.

4.3 Stream Reasoning

DubExtensions is focused on processing data from multiple heterogeneous sources in the very open domain of smarter cities. A usual source of information in this domain is Linked Data, coming with RDF-based descriptions to enable automated inferences that, in their turn, allow performing queries even if we do not know the exact schema of the data. Because typically we do not know the content of a stream in advance, that capability becomes crucial.

This section details *InfMod*, an optimized reasoning module for the RDFS, the most commonly used logic in Linked Data that consists of some basic vocabularies and a set of rules. These rules are applied to the input so as to derive additional information. For example, by using an algorithm such as RETE [24], it is possible to apply each rule recursively until no new input can be derived. In the relevant literature, this is called materialization, forward chaining, or forward reasoning, and it is well-suited for situations where reads dominate writes (since it greatly simplifies data queries).

An alternative approach, typically called backwards chaining or goal-driven reasoning, is to start from a query and reason only over the data relevant for that query. This approach is generally better suited when the data changes often and when the reasoning process is contained to small subset of the data.

Unfortunately, neither approach is well suited for stream processing. Forward chaining seems an unnatural choice in a domain where queries are usually known in advance, such as stream processing, since it will materialize information that is known not to contribute to query results. Backward chaining poses a challenge with regard to deciding *what* is relevant for the reasoning process: A significant performance advantage of stream processing systems is that they do not need to remember all of the information in a stream. A pure backward chaining approach nullifies this advantage since it requires access to arbitrary parts of the input, in a straightforward implementation. In addition, backward chaining for RDFS can not be implemented with a fixed network of operators.

To overcome those limitations, we propose a novel hybrid reasoning approach for streams. The essence of our approach is that we use forward chaining for streams while filtering the input according to possible derivations from a backward chaining algorithm. In addition, we materialize the static knowledge and only load the part that is relevant (i.e., the part that matches the input for some goal in the query) for the backwards part. In this way, we reduce the input size for the runtime reasoning component and we reduce the size of the joins and intermediate results.

We operate on a commonly used subset of RDFS rules in high-performance reasoning systems [25][1]. In addition, we exclude schema extensions on the streaming part, since we consider this risky in an environment where input comes from untrusted sources and can not be examined in advance.

Considering the RDFS rules, we have identified the full set of possible input patterns (Figure 5). Upper case letters represent variables (X), and lower case represent constants (a). This table does not consider all possible combinations: some of them have been discarded because they would not be meaningful for our ruleset (e.g. $< a,a,a >$), and others because they do not provide any conditions that are useful to simplify the dependencies graph (e.g. $< X,Y,Z >$). Starting from these combinations and considering all possible rule dependencies, we identified a backward chain for every combination.

Figure 4 shows an example for backward chaining, realized for the pattern "*<a,Type,Y>*", and the resulting set of filters for this specific goal. The figure represents the dependencies between rules. Rules at the leaves receive the input (after it has been filtered). The filters are obtained by the materialization of the generic constant "*a*".

By applying backward chaining on all other possible triple pattern combinations, we can obtain the possible filters for each of them (see Figure 5). These filters represent additional constraints on resources and properties on rules assumptions, and contribute in reducing the number of triples that every single rule can accept and need to calculate.

For example, considering the triple pattern "*:Student rdf:type ?x*", we can say that the only necessary conclusion that could be generated by RDFS *rule 2* to answer the query is generated from assumptions where *S* is identified with "*:Student*" (*S* = *:Student*).

[1] Although one could note that the RDFS closure is formally infinite, the usage of a limited subset of rules is prevalent in practical applications.

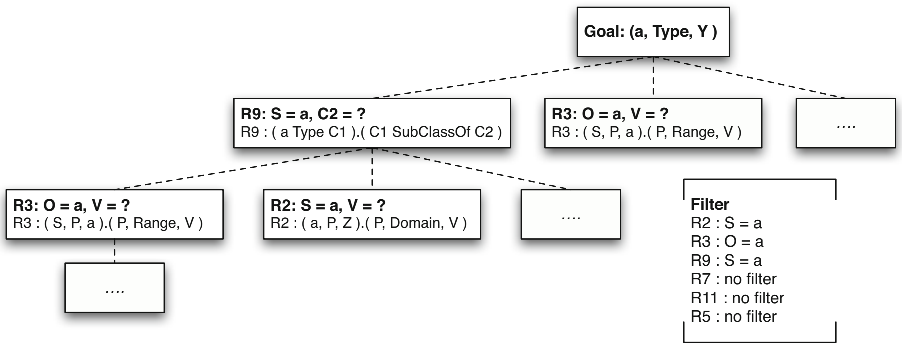


Fig. 4. Example of backward chaining

Combinations	R2	R3	R5	R9	R7	R11
< X, Type, Y >	✔	✔	✔	✔	✔	✔
< X, Type, c >	✔	✔	✔	C2 = c	✔	C3 = c
< a, Type, Y >	S = a	O = a	✔	S = a	✔	✔
< a, Type, c >	S = a	O = a	✔	S = a,C2 = c	✔	C3 = c
Combinations	**R2**	**R3**	**R5**	**R9**	**R7**	**R11**
< X, SubClassOf, Y >	✖	✖	✔	✖	✔	✔
< X, SubClassOf, c >	✖	✖	✔	✖	✔	C3 = c
< a, SubClassOf, Y >	✖	✖	✔	✖	✔	C1 = a
< a, SubClassOf, c >	✖	✖	✔	✖	✔	✔
Combinations	**R2**	**R3**	**R5**	**R9**	**R7**	**R11**
< X, SubPropertyOf, Y >	✖	✖	✔	✖	✔	✖
< X, SubPropertyOf, c >	✖	✖	✔	✖	✔	✖
< a, SubPropertyOf, Y >	✖	✖	✔	✖	✔	✖
< a, SubPropertyOf, c >	✖	✖	✔	✖	✔	✖
Combinations	**R2**	**R3**	**R5**	**R9**	**R7**	**R11**
< X, b, Y >	✖	✖	✔	✖	✔	✖
< X, b, b >	✖	✖	✔	✖	✔	✖
< b, b, Y >	✖	✖	✔	✖	✔	✖
< a, b, c >	✖	✖	✔	✖	✔	✖
Combinations	**R2**	**R3**	**R5**	**R9**	**R7**	**R11**
< a, X, Y >	S = a	O = a	✔	S = a	✔	✔
< X, Y, c >	✔	✔	✔	C2 = c	✔	C3 = c
< a, Y, Y >	S = a	O = a	✔	S = a, C2 = Type	✔	✔
< a, X, c >	S = a	O = a	✔	S = a,C2 = c	✔	✔
< X, X, c >	✖	✖	✔	✖	✔	✖

Fig. 5. Input filters for RDFS reasoning

Therefore, in our case, rule 2 can be rewritten as:

$$?P\ rdfs{:}domain\ ?V,\ {:}Student1\ ?P\ ?O \rightarrow {:}Student1\ rdf{:}type\ ?V$$

For the sake of completeness, Figure 5 shows the corresponding filters for all possible query pattern combinations. Finally, let us note that SPARQL queries normally contain multiple query patterns. Therefore, we can opt either to perform inferencing for each of them separately, or to configure filters to allow triples that match *any* of these patterns. Preliminary experiments have shown that, although the former option is more advantageous in terms of the number of triples that would be discarded, the latter option avoids duplicate computation that is frequent in our use cases.

5 Evaluation

Our language, methods, and system are validated through a set of complex queries, supporting the scenario described in Section 3 with real and heterogeneous data sources from Dublin City. The performance of our reasoning methods is evaluated through the widely-used LUBM reasoning benchmark [26], adapted for stream reasoning.

5.1 Language, Methods and System Validation

The sources of information we have deployed our system on are listed in Table 1. They include Linked Open Data, static data and stream data from sensors in Dublin (e.g., Dublin Bus locations and Dublin City noise data [27]) .

A set of queries, using the language in Section 4.1, with all our extensions such as the ones to specify different window time-spans, has been developed to capture the citizen quality of life scenario through KPIs and the CCTV camera selection (Section 3) scenario. The definition and (lengthy) description of the various KPIs and the camera selection queries, which go beyond the scope of this paper, are reported in the project home page[2]. The language features, the corresponding business logic, and the computational properties of our queries are reported in the remainder of this section.

For the sake of clarity and space, we illustrate our approach on the camera selection case, that is a complex case of KPI calculation (more complex queries and tighter time constraints). The camera selection scenario requires complex *filtering* and *aggregation*, namely, aggregation of camera based on KPI values and changes in sensor readings. The values of these indicators are weighted according to the distance to the closest camera, and then aggregated. We perform ranking based on a scoring formula and select the "best" cameras to observe at regular time intervals (i.e., every 2 seconds) by using the following main operations.

- **Aggregation.** Cameras are selected based on changes detected in the environment (through data in Table 1) such as noise, pollution, and buses in congestion. The camera selection required 40 group, 8 sum, and 38 average operations on data with different windows and source streams.
- **Reasoning.** Reasoning is mainly performed on categories of points of interest, grouped and described using subclass, subproperty, and type features of RDFS. Reasoning performance is thoroughly evaluated in Section 5.2.
- **Filtering & Ordering.** Filtering, ordering, and sorting has been applied for (i) selecting objects based on their spatial proximity and (ii) removing "NaN" values.
- **Time Slicing.** Multiple time windows have been considered to detect temporal changes in short- and long-term. For instance, in case a sensor reading significantly differs from the average, a higher score is assigned to it. Time slicing has been implemented through subqueries: 44 subqueries are considered in the camera selection scenario, with 44 different inputs/windowing configurations.
- **Distance Measurements.** Often, we need to calculate euclidian distances between geographic features. No specialized indexing structure has been used for calculating

[2] `http://www.lia.deis.unibo.it/Research/DubExtensions/`

Table 1. Datasets. A '*' denotes a dataset containing streaming information. When available, we are making references to publicly available fragments.

Dataset	**Description**	**Ref.**
Dublin Bus*	Position, identifier, and congested state of buses.	N/A
Pollution*	Ambient air quality, at a set of monitoring sites.	[28]
Ambient Noise*	Ambient noise, at a set of monitoring sites.	[27]
Points of Interest	4000 hierarchically-organized points of interest from LinkedGeoData using a radius of 15km around the centre of Dublin.	[6]
Weather	Weather reports, in CSV format extracted from a Web Service.	[5]
Pedestrian counts*	Readings from a set of footfall sensors in Dublin City.	N/A
Cameras*	Location of traffic monitoring and public safety CCTV cameras.	N/A
Dublin bikes*	Bikes, and bike slots available for bike-sharing scheme, per station.	[4]

distances. However, as the number of results can be large, we employ cardinal products and filtering for reducing the number of output results.

• **Joins.** Datasets are linked through joins. SPARQL is known to result in large numbers of join. The query for the camera selection scenario consists of 142 joins (not including the joins required for reasoning). However these joins are not performed on very large inputs, since the windows we are using are of limited size.

Using our system, we have realized a geographic information system viewer demonstrator that visualizes real-time streaming information about Dublin based on data streams in Table 1. A video, running on historical data for a single day is available on the Web[2].

Our language extensions and execution engine drastically reduce the development effort and time needed to write queries in the context of our scenarios. IBM Infosphere Streams already supports a concise language to define networks of processing operators (SPL). For our approach, combining the reasoning capabilities and our query language, we have replaced 7500 lines of SPL code with a query of only 512 lines (for the camera selection query). Our query was compiled to an equivalent SPL program, so performance was the same.

On-the-field performance results, processed on a virtual machine equipped with 10GB of RAM and two cores, confirmed that our system comfortably processes all streams in real-time, allowing the camera selection to take place every 2 seconds. Moreover, to stress-test the robustness of our system with regard to performance, we used stream logs to artificially increase the stream data input rate. Even with this limited deployment and resources, our system was able to cope with input rates 20 times faster than the actual streams in Dublin.

5.2 Performance Evaluation

Our second set of experiments evaluates the performance of our system in isolation, and also focuses on additional features of RDFS inference. Unfortunately, at the current stage, there are no benchmarks for RDFS reasoning over streams available in the literature. Thus, we have opted to evaluate our system against the LUBM reasoning benchmark [26], adapted for supporting temporal and stream reasoning aspects.

Both the *data-driven* and the optimized *hybrid* reasoning modes have been tested. In particular, we have measured performance in terms of maximum throughput, defined as the maximum number of triples processed per second without dropping messages due to buffer overflows. In addition, for the reasoning process, we have measured the latency, defined as the time between receiving an input and generating the corresponding output.

Dataset. Since our implementation can easily cope with the data from Dublin, we decided to use LUBM as a stress test for our system. LUBM has been developed to benchmark the inferencing capabilities of RDF stores: it models a university domain and allows testing various features of RDFS and OWL. For the purposes of our benchmark, we have split the LUBM benchmark in a static and and a dynamic part: we have considered information related to universities, professors, researchers, departments, and courses as static information, and all information related to students, such as supervisors, classes taken, etc., as dynamic information. The motivation behind this was to maximize the overlap between processing of static and dynamic information while maintaining a realistic data distribution.

Table 2. LUBM query set features

Query feature	**Q1**	**Q2**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q14**
Large Input	X			X	X	X	X	X	X
High Selectivity	X		X				X	X	
Low Selectivity				X		X			X
Querying only (not reasoning)	X								
Triangular Patterns		X					X		
Wide Hierarchies			X				X	X	
Several classes & Properties		X			X	X	X		
Reasoning on implicit relationships			X	X	X	X	X	X	
Reasoning on explicit relationships			X	X	X	X	X	X	
Query streaming and static knowledge		X	X		X	X	X		
Reasoning on streaming knowledge		X	X	X	X	X	X	X	X
Reasoning on static knowledge		X	X		X	X	X		

Query set. Without changing the semantics of the queries (apart from introducing windows for the streamed data), we modified the LUBM query set to work on streaming data. Because our system supports only RDFS-based reasoning, we have excluded the queries that require OWL reasoning or operate on static data only (Q3, Q4, Q11, Q12 and Q13). The full query set can be found online[2]. Table 2 lists key characteristics of the queries considered in our evaluation, derived from the LUBM query characteristics.

Experimental setting and Results. Our benchmark was run on an IBM blade server with a single Intel(R) Xeon(R) X5690 processor (6 cores, 3.47GHz) and 96GB of RAM, with Linux (RHEL 4.1.2-52) and InfoSphere Streams 2.0.0.3. Similarly to the experiments for Dublin City, to test scalability we have replayed the input while increasing the input rate until our system would produce either incorrect results or would drop messages due to buffer overflows.

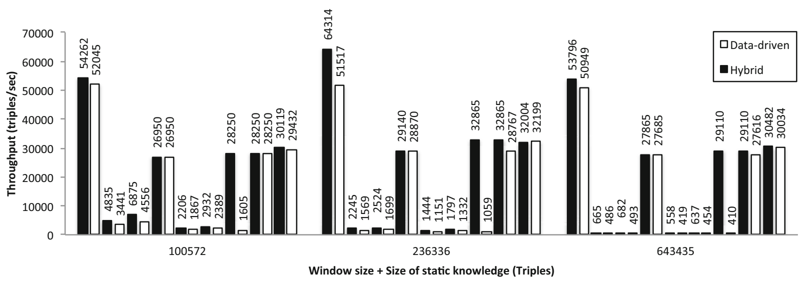


Fig. 6. Throughput as a function of window and static knowledge size

We have validated the correctness of our results against the Jena RDF store, with manually loaded data (for each window). Our system produced correct output for all results presented here. All results presented were obtained using Infosphere Streams.

For a combination of window size and static knowledge size, we have obtained the results shown in Figure 6. Each couple of columns represents one of our nine queries, in arithmetic order, (see Table 2) evaluated using hybrid and data-driven reasoning:

– For Q1, Q6, Q10 and Q14, hybrid and data-driven reasoning produce similar results. We attribute this to the fact that the input matches the filters presented in Section 4.3, so it is not possible to exclude much of the input early on. In addition, these queries are generally easier, since they required fewer joins and either no reasoning (Q1, querying only) or reasoning only on the stream.
– For queries combining both streaming and static knowledge (Q2, Q5, Q7, and Q8) we generally get worse performance because joins between stream and static knowledge have a higher hit rate.
– Hybrid reasoning usually outperforms data-driven reasoning, but data-driven reasoning never outperforms hybrid reasoning, that is due to fact that hybrid reasoning filters out information in the stream sooner (through the filters). In some queries, this difference is very significant: for Q9, hybrid reasoning is 17.6 times faster than data-driven reasoning.

To assess the scalability of our system, we repeated the tests with increasing window and static knowledge size. We see no significant drop in performance for easy queries (Q1, Q6, Q10 and Q14), while for other queries, we have:

– For hybrid reasoning, increasing the size of the window and the static knowledge by 2.4 and 6.4 times yields a decrease in average throughput by, respectively, 0.09 and 0.3 times, thus showing good scalability properties.
– For data-driven reasoning, instead, the same increase produces a throughput decrease of 0.51 and 0.84 times respectively, markedly higher than the hybrid reasoning one, but still not proportional to the input size.

We used the hybrid reasoner that showed the best performance in our previous tests to assess how window size influences the throughput when keeping the size of the static knowledge constant to 634,000 triples. Figure 7 shows obtained results with an increasing window size that ranges from 6,677 to 667,740 triples. This increase has relatively low impact on throughput, averaged over all queries; in fact, increasing the size of the window by a factor of [5, 10, 20, and 100] decreases the throughput by relatively small percentages, respectively [0.23, 0.33, 0.44, 0.58], demonstrating robustness and scalability. However, there is significant variation between queries: Q5 shows constant throughput, even for the largest (100x) increase, while Q9 shows the worst performance. We attribute these differences to the different number of streamed triples and the different complexity of the queries. In any case, the average decrease in throughput for all queries is better than proportional to the increase in window size.

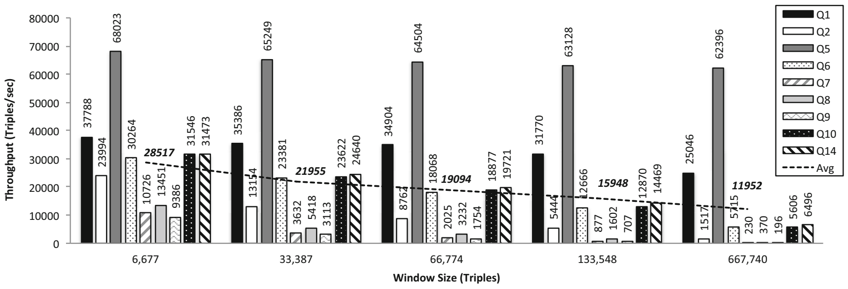


Fig. 7. Throughput as a function of window size

We collected latency results for both data-driven and hybrid reasoning, not shown here for brevity. All measurements show very small latency for query results, ranging from a few μs to less than 3ms, for both approaches. The only observed difference was that the hybrid reasoning is marginally faster than data-driven reasoning approach.

We have shown that the hybrid method presented in Section 4.3 significantly boosts the performance of stream reasoning. In addition, we have extensively studied the performance characteristics of our system, so as to give an indication of how our system would perform in even more challenging settings. Compared to other systems that perform stream querying and some that support stream reasoning over RDF data, such as the ones described in Section 2, our contribution is original in dealing with heterogeneous input, supporting hybrid inference on streams, and being deployable on a distributed clustered architecture. Unfortunately, given that this field is still in its infancy, there is still no agreement with regard to language specifications, logics employed, and semantics. This makes a direct comparison meaningless. Recent efforts toward a common benchmark [29] will hopefully allow more thorough comparison in the near future.

6 Conclusions

In this paper, we have presented an approach for real-time urban monitoring and reasoning by using a real-world scenario for Dublin City. Besides the scenario itself, the contribution of this paper lies in a set of extensions to current stream reasoning languages to tackle heterogeneity and support our use-cases. Furthermore, we have outlined a stream reasoner based on an industry-strength engine, IBM Infosphere Streams.

Window Size	Q1	Q2	Q5	Q6	Q7	Q8	Q9	Q10	Q14	Avg
6,677	37788	23994	68023	30264	10726	13451	9386	31546	31473	28517
33,387	35386	13154	65249	23381	3632	5418	3113	23622	24640	21955
66,774	34904	8762	64504	18068	2025	3232	1754	18877	19721	19094
133,548	31770	5444	63128	12666	877	1602	707	12870	14469	15948
667,740	25046	1517	62396	5715	230	370	196	5606	6496	11952

Finally, we have presented an efficient RDFS reasoning approach for streams and have evaluated our system using real data that shows very low response times.

Future directions lie in extending our stream processor to deal efficiently with space, due to its importance in an urban context. In addition, a critical point for future stream reasoning systems will be robustness. In an open setting, further investigation is due to the *failure models* for distributed streams. Finally, parallel reasoning is a very promising area. Efficiently distributing stream computation across the nodes of a cluster has the potential to scale to much larger problem sizes.

References

1. Valle, E.D., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems 24(6), 83–89 (2009)
2. Babu, S., Widom, J.: Continuous queries over data streams. SIGMOD Record 30(3), 109–120 (2001)
3. Shadbolt, N., O'Hara, K., Berners-Lee, T., Gibbins, N., Glaser, H., Hall, W., Schraefel, M.: Linked open government data: Lessons from data.gov.uk. IEEE Intelligent Systems 27(3), 16–24 (2012)
4. Nash, O.: Dublin bikes revisited (blog post), `http://ocfnash.wordpress.com/2011/02/02/dublin-bikes-revisited/`
5. Weather Underground Web Service, `http://www.wunderground.com/history/`
6. Stadler, C., Lehmann, J., Höffner, K., Auer, S.: Linkedgeodata: A core for a web of spatial open data. Semantic Web (2011)
7. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A nucleus for a web of open data. In: Aberer, K., et al. (eds.) ISWC/ASWC 2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007)
8. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: Ep-sparql: a unified language for event processing and stream reasoning. In: WWW, pp. 635–644 (2011)
9. Le-Phuoc, D., Dao-Tran, M., Parreira, J. X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
10. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: C-sparql: Sparql for continuous querying. In: WWW, pp. 1061–1062 (2009)
11. Biem, A., Bouillet, E., Feng, H., Ranganathan, A., Riabov, A., Verscheure, O., Koutsopoulos, H.N., Moran, C.: IBM infosphere streams for scalable, real-time, intelligent transportation services. In: SIGMOD, pp. 1093–1104 (2010)
12. Luo, C., Thakkar, H., Wang, H., Zaniolo, C.: A native extension of sql for mining data streams. In: SIGMOD, pp. 873–875 (2005)
13. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring streams - a new class of data management applications. In: VLDB, pp. 215–226 (2002)
14. Liu, L., Pu, C., Tang, W.: Correction to "continual queries for internet scale event-driven information delivery". IEEE Trans. Knowl. Data Eng. 12(5), 861 (2000)
15. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Reiss, F., Shah, M.A.: Telegraphcq: Continuous dataflow processing. In: SIGMOD Conference, p. 668 (2003)

16. Bolles, A., Grawunder, M., Jacobi, J.: Streaming SPARQL - extending SPARQL to process data streams. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 448–462. Springer, Heidelberg (2008)
17. Rodriguez, A., McGrath, R.E., Liu, Y., Myers, J.D.: Semantic management of streaming data. In: International Workshop on Semantic Sensor Networks at ISWC (2009)
18. Ren, Y., Pan, J.Z.: Optimising ontology stream reasoning with truth maintenance system. In: CIKM, pp. 831–836 (2011)
19. Sandu, D.: Operational and real-time business intelligence. Revista Informatica Economică 3(47), 33–36 (2008)
20. Eckerson, W.W.: Performance dashboards: measuring, monitoring, and managing your business, 2nd edn. John Wiley and Sons (2010)
21. Pankanti, S., Brown, L., Connell, J., Datta, A., Fan, Q., Feris, R., Haas, N., Li, Y., Ratha, N., Trinh, H.: Practical computer vision: Example techniques and challenges. IBM Journal of Research and Development 55(5), 3:1–3:12 (2011)
22. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
23. IBM: Ibm infosphere streams - harnessing data in motion. In: WWW (2010)
24. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1), 17–37 (1982)
25. Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., Bal, H.: Webpie: A web-scale parallel inference engine using mapreduce. Web Semant. 10, 59–75 (2012)
26. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics 3, 158–182 (2005)
27. Dublin City Council: Noise monitoring data, `http://dublinked.ie/datastore/datasets/dataset-142.php`
28. Dublin City Council: Air pollution monitoring data, `http://dublinked.ie/datastore/datasets/dataset-185.php`
29. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.-P.: SRBench: A streaming RDF/SPARQL benchmark. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 641–657. Springer, Heidelberg (2012)