ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTA' DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

DIPARTIMENTO DI ELETTRONICA INFORMATICA E SISTEMISTICA

TESI DI LAUREA

in
Reti Di Calcolatori M

Processing di Flussi di Dati Eterogenei per Smart City
(Heterogeneous Data Stream Processing For Smart Cities)

CANDIDATO
Simone Tallevi-Diotallevi

RELATORE:
Chiar.mo Prof.
Antonio Corradi
CORRELATORI:
Dott. Spyros Kotoulas
Dott. Pol Mac Aonghusa

Anno Accademico 2011/12

Sessione II

CONTENTS

LIST OF FIGURES **5**

LIST OF TABLES **9**

INTRODUCTION **11**

1 BASIC CONCEPTS **13**

1.1 SEMANTIC WEB 13

1.1.1 Linked Data *14*

1.1.2 Ontology *15*

1.1.3 Query engine *17*

1.2 DATA STREAMS 18

1.2.1 Application Domains *19*

1.2.2 Access data model *20*

1.2.3 The concept of window *22*

1.2.4 Streaming Linked Data *23*

1.2.5 Difference between Knowledge and Data *24*

2 BACKGROUND **27**

2.1 SEMANTIC WEB TECHNOLOGIES 27

2.1.1 Resource Description Framework *28*

2.1.1.1 RDF data model 29

2.1.1.2 Serialization formats 30

2.1.2 RDF vocabulary description language (RDF Schema) *32*

2.1.2.1 RDFS classes and properties 33

2.1.2.2 Web Ontology Language (OWL) 36

2.1.2.3 Ontology Hijacking 36

2.1.3 RDF Schema Entailment Rules *38*

2.1.4 SPARQL Protocol and RDF Query Language *40*

2.1.4.1 SPARQL and Structured Query Language 41

2.1.4.2 SPARQL query structure 42

2.1.4.3 SPARQL graph patterns 45

2.2 DATA STREAM PROCESSING 47

2.2.1 Data Stream Management System architectural model *48*

2.2.2 Continuous Query *49*

2.2.3 Existing Data Stream Management Systems *51*

2.2.3.1 Stanford stream data manager ... 51
2.2.3.2 IBM InfoSphere Streams ... 52
2.3 STREAMING LINKED DATA PROCESSING ... 53
2.3.1 The window configurations ... 55
2.3.2 Stream Reasoning ... 56
2.4 EXISTING STREAMING LINKED DATA PROCESSORS ... 58
2.4.1 Querying Streaming Linked Data with DSMS and SPARQL ... 59
2.4.2 A native and adaptive query processor for Streaming Linked Data ... 60
2.5 INFOSPHERE STREAMS AND STREAMS PROCESSING LANGUAGE ... 62
2.6 ARQ ENGINE OF JENA FRAMEWORK ... 64
3 THE NEW SPARQL EXTENSION DEFINITION ... 67
3.1 QCELS ... 68
3.2 C-SPARQL ... 70
3.3 SPARQL FOR HETEROGENEOUS DATA STREAMS ... 72
3.3.1 Analysis and comparison ... 73
3.3.2 New SPARQL extension definition ... 76
4 DUBEXTENSIONS EXECUTION FRAMEWORK ... 81
4.1 STREAMS PROCESSING LANGUAGE OPERATORS ... 81
4.1.1 Relational Operators ... 83
4.1.1.1 SPL stateless operators ... 83
4.1.1.2 SPL stateful operators ... 84
4.1.2 Adapter Operators ... 88
4.1.3 Utility Operators ... 88
4.2 LOGICAL TRANSLATION OF DUBEXTENSIONS QUERY TO SPL ... 89
4.2.1 First step: the ARQ translation ... 90
4.2.1.1 Standard SSE translation of a SPARQL query ... 91
4.2.1.2 SSE translation of a DubExtensions query ... 96
4.2.2 Second phase: the SPL translation ... 98
4.2.2.1 Input specifications module ... 99
4.2.2.2 Logical evaluation module ... 100
4.2.2.3 Window configurations ... 108
4.2.3 A complete translation example ... 112
5 BACKWARD STREAM REASONING ... 115
5.1 RULES DEPENDENCIES ... 115
5.1.1 Triple pattern combinations ... 118
5.1.2 Backward reasoning filters ... 121
5.2 STATIC BACKWARD REASONER ... 124

5.2.1 The rules connections ... *125*
5.2.2 Output filters configurations ... *126*
5.2.3 Complete static backward reasoning module ... *126*
5.3 STREAM BACKWARD REASONING ... 127
5.3.1 Streaming component ... *129*
5.3.2 Complete backward stream reasoning module ... *131*
5.4 REASONING MODULES IN STREAMING APPLICATIONS ... 132
6 REAL USE CASE AND PRELIMINARY PERFORMANCE ... 135
6.1 PERFORMANCE EVALUATION ... 135
6.1.1 LUMB queries ... *136*
6.1.2 Preliminary performance ... *140*
6.1.2.1 Forward/Backward Reasoning ... 140
6.1.2.2 Backward Stream Reasoning on different window sizes ... 143
6.2 DUBLIN CITY MONITORING ... 145
6.2.1 Query and dataset definition ... *145*
6.2.2 Simulation ... *151*
6.3 CONCLUSIONS AND FUTURE WORKS ... 153
CONCLUSION ... 157
REFERENCES ... 159

List of Figures

Figure 1.1 Linked Open Data cloud diagram [7]..15
Figure 1.2 Hierarchical information of the Person concept......................................16
Figure 1.3 A logical representation of a data stream [11] 18
Figure 1.4 Pull-based data access model 21
Figure 1.5 Publish/Subscribe push-based model 22
Figure 1.6 Logical example of Window on Data Stream [11]..................................23
Figure 2.1 Logical representation of RDF statements 29
Figure 2.2 Example of RDF dataset as a directed graph [20]...................................30
Figure 2.3 Representative set of RDF serialization formats.....................................31
Figure 2.4 Example of *Ontology Hijacking*..37
Figure 2.5 Example of correct ontology extension...37
Figure 2.6 Example of SQL query behaviour..41
Figure 2.7 Example of SPARQL query behaviour 42
Figure 2.8 A simple SPARQL query 44
Figure 2.9 SPARQL query with multiple triple patterns 47
Figure 2.10 DSMS architectural model 48
Figure 2.11 Logical representation of a streaming application 50
Figure 2.12 Example of Continuous Query Language query [39] 51
Figure 2.13 Model of STREAM continuous queries [39] 52
Figure 2.14 Sliding and tumbling time-based window...56
Figure 2.15 Window on RDF stream..57
Figure 2.16 Stream reasoning – RDFS rule 2 57
Figure 2.17 C-SPARQL architecture overview 59
Figure 2.18 CQELS processing model 61
Figure 2.19 SPL Code and graphical representation [46]...63
Figure 2.20 ARQ architecture overview [20] 64
Figure 3.1 CQELS SPARQL extension definition [44] 68
Figure 3.2 CQELS stream graph pattern examples 69
Figure 3.3 Example of CQELS query [44] 69
Figure 3.4 C-SPARQL SPARQL extension definition [13].....................................70

Figure 3.5 Example of C-SPARQL query [13] 71
Figure 3.6 Comparison between C-SPARQL and CQELS extensions [44,35]........ 75
Figure 3.7 *DubExtensions* SPARQL extension definitions 76
Figure 3.8 *DubExtensions* stream and CSV graph pattern examples 78
Figure 3.9 *DubExtensions* input specifications... 79
Figure 3.10 Example of *DubExtensions* query 79
Figure 4.1 Generic SPL stream node [41] 82
Figure 4.2 Example of SPL Filter operator.. 83
Figure 4.3 Example of SPL Functor and Punctor operators 84
Figure 4.4 Example of SPL Sort operator.. 85
Figure 4.5 Example of SPL Join operator (*inner join*) 85
Figure 4.6 Example of SPL Join operator (*Cartesian Product*) 86
Figure 4.7 Example of Left/Right/Outer Join 87
Figure 4.8 Example of SPL Aggregate operator.. 87
Figure 4.9 Example of SPL FileSource and FileSink operators 88
Figure 4.10 Example of Custom operator.. 89
Figure 4.11 Example of Beacon operator 89
Figure 4.12 Example of binary operator performing a Join on three BGP.............. 92
Figure 4.13 Example of complete translation of a SPARQL query to SS-Expression 95
Figure 4.14 Complete example of DubExtensions first phase translation................ 97
Figure 4.15 Overview of a SPL streaming application for a DubExtensions query. 98
Figure 4.16 Logical representation of the Parser node and its tuples results............ 99
Figure 4.17 Auxiliary operator Split node 101
Figure 4.18 Example of outside Split node routing 102
Figure 4.19 Auxiliary operator Union node... 102
Figure 4.20 Example of composite Standard BGP operator.................................. 103
Figure 4.21 Algorithm used to find all possible matches 104
Figure 4.22 Example of transformation from a BGP CSV to stream schema 105
Figure 4.23 Example of CSV BGP single operator 106
Figure 4.24 Example of project and extend single operator 106
Figure 4.25 Example of filter single operator.. 106
Figure 4.26 Example of sort single operator.. 106
Figure 4.27 Example of group single operator 107

Figure 4.28 Example of slice single operator 107
Figure 4.29 Example of Join/LeftJoin single operator translation..........................108
Figure 4.30 Example of Window configuration applied to a stream......................109
Figure 4.31 Biggest and Smallest output window configurations 110
Figure 4.32 Example of problem lead by the buffer nodes 111
Figure 4.33 Example of Window configuration in a Bugger node.........................111
Figure 4.34 Two level window configurations..112
Figure 4.35 Example of DubExtensions query..113
Figure 4.36 Example of DubExtensions query, after the first translation phase 113
Figure 4.37 Example of DubExtensions query, after the second translation phase.114
Figure 5.1 Dependency graph of relationships between the subset of RDFS rules [30]...116
Figure 5.2 Dependency graph with explicit relationships 117
Figure 5.3 Combinations of all possible triples patterns 118
Figure 5.4 Correspondence triple pattern with triple combination.........................119
Figure 5.5 Dependency graph of the triple pattern combination *X SubClassOf c* ...119
Figure 5.6 Backward chaining on "*a Type Y*" combination 122
Figure 5.7 Example of merge of two triple pattern combinations..........................123
Figure 5.8 Relationships between Backward and Forward Reasoning 124
Figure 5.9 Example of Join node configuration to implement Rule2.....................125
Figure 5.10 Split node configuration in the static reasoning module 125
Figure 5.11 Logical representation of the static stream reasoning module 127
Figure 5.12 Overview of the Stream reasoning module 129
Figure 5.13 Configuration rules in the Streaming component...............................130
Figure 5.14 Logical Architecture overview of the Stream reasoning module.........131
Figure 5.15 Complete overview of a Streaming application with reasoning modules 132
Figure 5.16 Functionality of FROM ONTOLOGY clause.....................................133
Figure 6.1 LUMB Query 2 expressed in DubExtensions 138
Figure 6.2 Forward/Backward reasoning approach...141
Figure 6.3 Forward/Backward reasoning approach (first iteration) 142
Figure 6.4 Backward stream reasoning performance on different window sizes....144
Figure 6.5 Section of the hierarchical organisation of the amenities of Dublin 148
Figure 6.6 The KPI query results clause...148

Figure 6.7 The KPI query dataset definition... 148
Figure 6.8 the KPI query pattern - selection of the camera information weighted by the number of near amenities... 149
Figure 6.9 the KPI query pattern - selection of the average of pollution level...... 149
Figure 6.10 Graphical representation of the KPI DubExtensions query to monitor Dublin city 150
Figure 6.11 Screenshot of Dublin city simulation 152

List of Tables

Table 2.1 Example of RDF dataset [20] 30
Table 2.2 Example of RDF and RDFS dataset 35
Table 2.3 RDF Schema Entailment Rules [29]....................40
Table 2.4 RDF dataset – Library 46
Table 4.1 Example of SSE Basic Graph Pattern translation....................92
Table 4.2 Example of SSE Group Graph Patter translation 93
Table 4.3 Example of SSE Optional Graph Patter translation....................93
Table 4.4 Example of SSE Optional Graph Patter translation....................94
Table 4.5 Example of SSE filter operator....................94
Table 4.6 Example of SSE slice/order/group operators....................95
Table 4.7 Generic RDF, RDF stream and CSV stream specifications 100
Table 5.1 Subset of RDFS Entailment rules....................117
Table 5.2 Summary table of subset of rules for every combination....................120
Table 5.3 Summary table of all possible filters applicable to all possible triple combinations....................122
Table 5.4 The subset of rules exploit by the streaming component 128
Table 6.1 Subset of LUMB query and relative classification [50]....................138
Table 6.2 LUMB queries results with OWL rules and with RDFS rules 139
Table 6.3 Dataset configurations 140
Table 6.4 Complete dataset of the DubExtensions monitoring query 147

INTRODUCTION

The continued urbanization is an unstoppable trend, which drives existing cities to enlarge themself and become bigger, increasing simultaneously their scalability problems. In this extension trend, we need to find a way to manage this novel complexity to increase efficiency and to reduce expenses of a smart and sustainable city, to improve the quality of life of its citizens. Since our cities are becoming bigger and bigger, they need to become even smarter.

Smarter, in a city context, means to be able to monitor, analyse, and foresee city behaviours by interpreting information and consequentially, to provide better services and an easier and safer life to citizens. Smart city is a new area of interest for both business companies and research institutions all over the world, with a goal of devising and realising next-generation cities. This new concept of city starts from the assumption that everything is usable to provide information: citizen mobiles, traffic sensors, monitoring stations, public transportation, and so on; all those can be integrated to provide complete description of city environments, to foresee new tendencies and even influence future behaviours. In this way, a smart city can be considered like a collection of real-time data flows from many possible sources that need to be put together for better services for citizens.

However, the information of the city is no longer limited to sensors, but in a more integrated vision, open to citizens: by exploiting the power of integration of the Web, citizens can be able to share real-time information about every possible topic; traffic conditions, social events, and also open problems, can be fatherly used to understand which parts of the city provide a better quality of life, and where the citizens are happier.

Our project is along this prospective, whose goal is to provide a way to merge all these real-time heterogeneous information together, for better and more complete services, to transform our cities into smarter ones. This kind of integration is far from trivial, the solution must work on real-time information, even if they are still strongly related to specific data- representation formats. We have designed a new system able to overcome such limitation and to satisfy specific requests by processing real-time heterogeneous stream data from the city.

This work have been realised in collaboration with IBM Research Dublin and it is part of a larger project of research on Smarter Cities, which is currently conducting research in water, energy, transportation, city fabric, risk, exascale computing, and marine environments [1], considering the real-time environment of Dublin city.

This dissertation we will pass through every single step that we have exploited in order to realise our system. We will start to give some basic concepts of Data Stream and Semantic Web (Chapter 1), and then we will go through their specific technologies, providing also an overview of existing systems (Chapter 2). Moreover, we will present a formal definition of our new language to realise requests on heterogeneous data streams (Chapter 3), and then their necessary translations, in order to be executed on Data Stream Processing systems (Chapter 4). We also improve our system by providing additional reasoning capabilities, exploring backward stream reasoning techniques (Chapter 5). The last chapter of this dissertation will evaluate our system by considering performance capabilities of our stream reasoning approach, and by exploiting our system in a real use case with Dublin city (Chapter 6).

1 BASIC CONCEPTS

We are living into an era where the development of information and telecommunication systems is continuously evolving. This has led to incredible changes into the human life. This development is still underway and proposes an innumerable amount of data with several organizations.

Flows of data (*Data streams*) represent a new kind of dynamic and time-dependent information that describes real-time environments, where data come from different sensors, continuously, and in real-time.

These real-time data are becoming more often available on Internet sites, where they are shared, distributed, and transformed in useful and user-available data. In this way, they are becoming a "dynamic extension" of the *World Wide Web (WWW* or *W3 or Web)* contents.

Since the new Web applications need to process this rich information, the Web needs to be extended to provide the semantic meaning of the data, so to become more useful and automatically interpretable by machines. In the same way as Web information, *Data Stream*s should be considered into the new semantic vision of the Web.

The next chapter will present the basic concept of both the *Semantic Web* and *Data Streams*.

1.1 SEMANTIC WEB

Everyone knows about the Web, a collection of Internet sites built for human consultation and consumption [2]. This collection of sites composes Web information that can be considered static, even if the contents of the Web pages change over a long period of time: they can be considered static for sure if compared to real-time information. Moreover, since the Web is built for human consumption, information, fully available to humans, is machine-readable but not machine-understandable: even if machines are able to read the information, they cannot understand the meaning of those data. In order to overcome such limitation it is needed to describe the existing information with other additional information

typically called *metadata* [3]. *Metadata* allow the typing of data to make content machine-understandable and they are used to provide *semantics* to information, in the sense of the interpretation of the meaning, of a word, sign, sentence, and so on. Thus the semantic of web information provides the meaning to web data and resources. It describes how these elements are related to each other, and how they are related to the real world objects.

By exploiting metadata a calculator becomes able to directly use and process web contents in agreement with the vision of the *Semantic Web*.

The *Semantic Web* represents the *W3 Consortium (W3C)* vision of the Web [4]. It is an extension of the existing World Wide Web that allows people and machines to understand the meaning of web information. It provides a standardised way for expressing relationships between data that are given in a common format. This means a better cooperation between people, but more important, between machines. In this scenario machines are able, exploiting the relations between data, to start off from a specific database and then pass to another one, following the connection between information with related meaning.

In the vision of the Semantic Web the Web becomes the *Web of Linked Data* [4], a collection of interconnected information that can be navigated by machines. In this way the Web becomes a network of machine-understandable concepts.

1.1.1 Linked Data

The *Web of Linked Data* are collections of linked information connected by metadata, so to produce a network on interconnection that can be navigated by machines (Figure 1.1).

Linked Data describe a method of publishing structured data so that they can be interlinked and become more useful, allowing large-scale integration of data on the Web [5]. Linked Data represent a radical change in the organisation of information, composed with *relational databases*. Relational databases exploit static and "flat" tables in order to store information that results scarcely connected [6]. On the other hand, Linked Data organise the information as a strictly connected graph (Figure 1.1), expressing the data in a more complete form. That can also be called *Knowledge Base (KB)*.

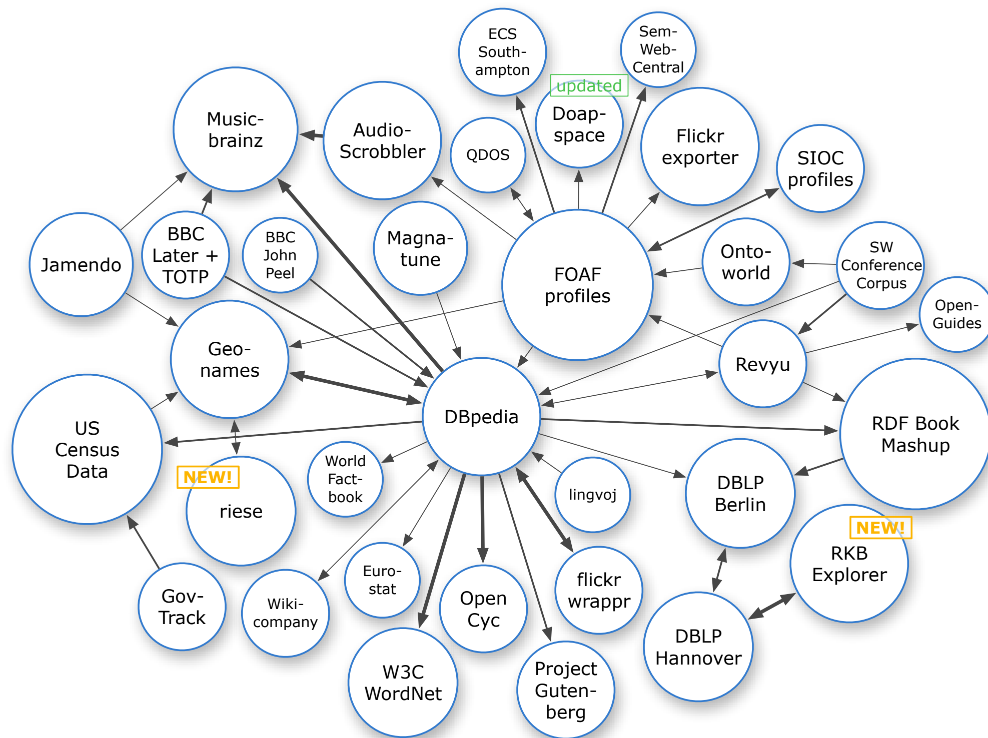


Figure 1.1 Linked Open Data cloud diagram **[7]**

The next subsection will present the concept of Ontology that describes how data have to be connected together.

1.1.2 Ontology

Exploiting Linked Data, as we have said, it is possible to navigate information based on their contents. However they are merely a way to describe and organise data on the Web, but they do not specify any description about how those data have to be linked together, and how some concepts are interconnected. This task is performed by *Ontology*.

Ontology is a term borrowed from philosophy that refers to the science of describing kinds of concepts in the world and how they relate [8]. In Semantic Web, an ontology is a formal representation of knowledge as a set of concepts within a specific domain, and also a description of how these concepts relate to each other. The relationship is the key element of ontology and it represents a connection between concepts.

For example, an ontology can be used to define the concepts of *Person* and *Student* and how they are related to each other: *A Student is also a Person.* In that way, an ontology defines a set of concepts that form *vocabularies of terms*. There are different vocabularies of terms for different application domains, because the same concept can change its meaning depending on the context.

In the Semantic Web, *Uniform Resource Identifiers (URIs)* identify the concepts, represented by Web resources, and vocabularies of terms are used to define resources and relationships between such resources. In this way, ontology can produce a description of how those resources are related. This description represents the structure *(*or *schema)* of web information, which gives the guidelines used by Linked Data to represents the information [5]. In the same way as web information, schema information is expressed as Linked Data.

In the example below one can see possible hierarchy of information that can be used to describe the *Person* concept.

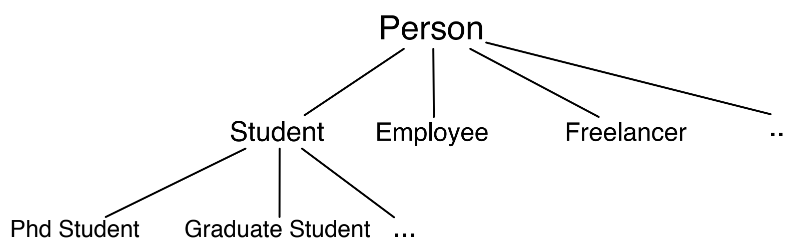


Figure 1.2 Hierarchical information of the Person concept

Ontology uses a set of default elements in order to describe relationships between data, and based on the number of these default elements, it is possible to say that a particular type of ontology is more expressive than another. It is then possible to describe more relationships between data, to produce more complete structure information.

Since Ontology is expressed as Linked Data, machines can automatically process them. However, in order to extract the meaning of the information, we need a *query engine* and a *reasoner*, which we will present in the next subsection.

1.1.3 Query engine

A *query engine* is a generic system able to process specific and concise requests called *queries* and it is used in both traditional database and knowledge base. It is also able to retrieve information expressed by Linked Data and described by ontologies, answering to specific queries. In order to work on KB a query engine exploits a *Rule-base Reasoner*, that is a system able to extract the semantic meaning of structured information and to compute complex reasoning over a KB [9], by exploiting the *logic programming* concepts of *Artificial Intelligence (AI)* systems, that stating from a set of assumptions, they exploit a set of inference rules to reach and demonstrate specific goals. In the same way, a Rule-Based Reasoner exploits a set of *Entailment (*or *Inference) Rules* to deal with Linked Data. Entailment Rules allow extracting the meaning of Linked Data and "understand" the information that is described by them. They are able to start from some specific assumptions, and infer multiple conclusions, which can also be new assumptions for other rules, and so on. This produces a *chain* of rules activations, which is used by the *Rule-base Reasoner* to exhaustively infer all needed *conclusions* of a specific KB. The *Rule-base Reasoner* can apply this chain of activations in two opposite ways [9]:

- *Forward chaining (forward reasoning)*: it is also called *materialization* since rules are applied over the entire KB until all possible information is "materialised" (derived from the existing knowledge). When the materialization is terminated, it means that the *complete materialization* or the *closure* is reached. The main advantage of this approach is that when the closure is calculated one can easily process the complete set of information. On the other hand, the main disadvantage is that the closure needs to be updated at every change in the KB.
- *Backward chaining (backward reasoning)*: the set of rules is applied starting from a goal only over the strict set of formulas that are necessary to prove it. Since reasoning is performed for a specific request, it is cheaper updating the knowledge base, then in the forward reasoning approach. However, since the complete materialization is not calculated, the system has to perform specific computations for every request.

If we consider again the hierarchical structure of the Person concept (Figure 1.2), a query engine without a Rules-based Reasoner cannot understands the relation

between the concept of *Person*, and the concept of *PhD Student,* unless it is not clearly defined. However the entailment rules enable that knowledge and allow the finding.

For example, if a user request is: *select all people*, but just *Student* and *PhD Student* are composing the KB, a query engine without reasoner will answer to the query with an empty response. On the other hand, a query engine that exploits a reasoner is able to find all possible results.

1.2 DATA STREAMS

Data streams are a potentially unlimited collection of data *(tuples)*, used to express real-time information [10]: they are extremely dynamic and time varying, because they describe real-time environments, where data are continuously arriving and the previous information is usually no longer necessary.

An example of this kind of environments is the new concept of city, often called smart city. In the smart city area, data come from real-time sensors by producing a stream of data, which are interesting just for a short period of time because we need to manage real-time information in order to solve abrupt problems.

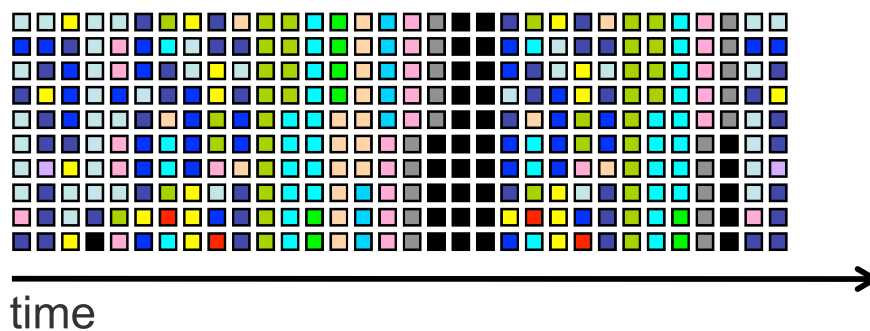


Figure 1.3 A logical representation of a data stream **[11]**

Data streams come from multiple different sources with heterogeneous data representations [10]: streams can be logical represented as an ordered sequence of couples, where each one is composed by a *tuple* and its *timestamp* (τ_i), as in the representation shown below. In this case the timestamp linked to the tuple expresses the time varying nature of the data [12].

$$< \text{tuple1}, \tau_1>$$
$$< \text{tuple2}, \tau_2>$$
$$< \text{tuple3}, \tau_3>$$

...

The sequences of timestamps are monotonically non-decreasing values in a stream, because values do not have to be unique [13]. Moreover since the information is coming from real-time sources and the amount of information cannot be store, it is not possible to select information of the past.

Data streams are characterised by:

- *large amount of information*: the amount of data stream is usually so high that is impossible to store it, therefore the information has to be consumed on the fly;
- *dynamic information*: since data are coming from real-time sources, information is continuously changing, and it is extremely dynamic;
- *real-time data*: the stream environments provide real-time data that have to be used in real-time. Information has to be collected and given to the user, as soon the request has been evaluated. The latency is essential working on data streams;
- *multiple sources*: different sources provide information with different frequency and every source could have heterogeneous formats.

The next subsection will present a sample of application-domains that exploit the characteristics of data streams.

1.2.1 Application Domains

The potential of data streams have defined new types of application-domains, where one needs to process large volumes of continuously arriving data with high throughput and low latency. These involve heterogeneous domains, arriving as well to our daily life, and producing incredibly changes. Examples of these application-domains are [14,15]:

- *smart cities*: they are real-time environments nearest to our daily life, they represent the collection of real-time information that can be obtained from bus and car positions in the city, pollution and noise level or the energy consumption of the city. All these data come from heterogeneous sensors, which together form a *sensor network* that constantly produces information as data streams. In this scenario new applications are used to monitor this kind of environment, and manage its complexity. Their goal is to increase the city services and in this way improve the quality of life for citizens;

- *network traffic managements*: they monitor and analyse streams of network packets on the fly, in order to find out information on traffic flow patterns. This type of information is really important in routing system analysis, bandwidth usage statistics, and network security, since it can be used to provide real-time responses against intrusion detection to prevent denial-of- service attacks;
- *computational finances*: they exploit the ability to process and analyse large amounts of financial data and extract useful information. These application-domains are used to detect advantageous market conditions and to execute rapid, automatics, and more convenient trades.

These application-domains that need to process large volumes of information, have forced to change data processing. Therefore it has forced to define a new paradigm of data processing: *data stream processing*. Able to work on data streams where data cannot be stored and they have to be consumed on the fly. Since data cannot be stored the traditional access data model is no longer valid and it has been defined a new data access model, which we will present in the next subsection.

1.2.2 Access data model

As we have already said, data stream processing is based on the assumption that data streams cannot be stored, since they are an unlimited amount of information, and they have to be consumed on the fly. In order to work on data streams, an access data model has to deal with these characteristics and provide correct answers to the requests.

A *client/server (C/S) pull-based* data access model (Figure 1.4) is an example of access data model used by traditional databases, where static not frequently changing data are saved on physical supports. In C/S pull-based model the clients (user-applications) play the active role, asking for the execution of a specific query and the server (Static Database) that plays the passive role, answering to such queries [10]. In Figure 1.4 one can see a logical representation of a pull-based model.

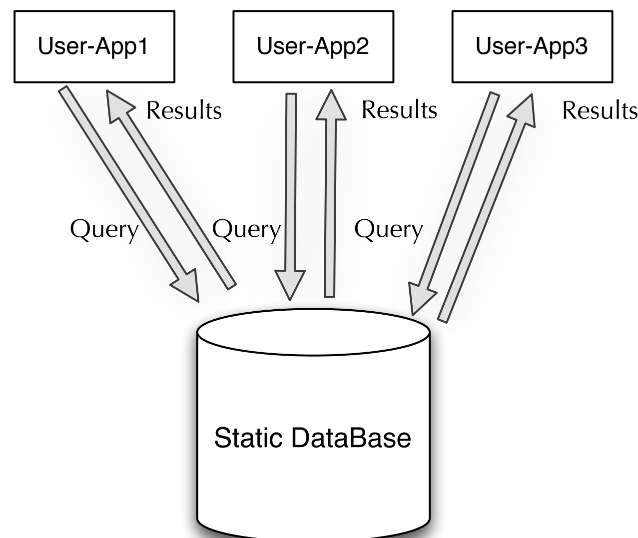


Figure 1.4 Pull-based data access model

However in order to work on data stream this traditional approach is no longer valid, since data cannot be stored and they have to be queried on the fly. In order to process flowing information, data stream processing systems exploit a new type of query that is able to run continuously over a period of time: *Continuous Queries (CQs)* [10]. Continuous Queries are registered into the systems that manage the stream and from that moment on, they start to execute on data streams. Moreover they change the access data model of user applications that need to pass from pull to *push-based model*. That happens because data are continuously coming and the results of the queries have to be continuously updated by pushing the values to the user-applications.

Tuples of data streams can be seen like events generated by multiple data stream sources, which are responsible to capture events of interest happening in a specific environment. In order to work with multiple sources, data stream processing systems provide an access data model, which guarantees a decoupling between continuous information sources and final users, exploiting a *publish/subscribe (pub/sub) push-based* model (Figure 1.5) [16]. In pub/sub push-based model there are three mains components, *publishers* (senders), *subscribers* (receivers), and a *message broker* (central exchange point). In a stream scenario, data stream sources are publishers, user-applications are subscribers, and the data stream processing system is the message broker [16].

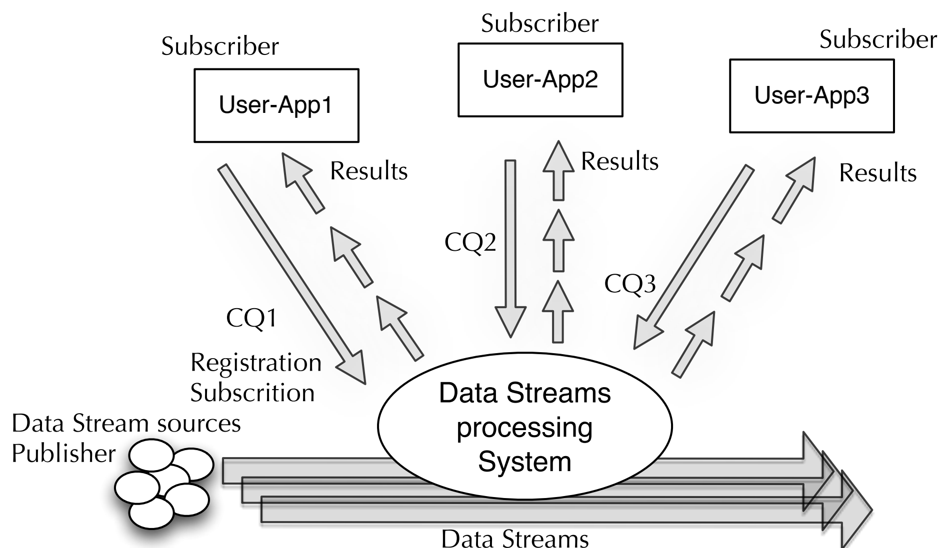


Figure 1.5 Publish/Subscribe push-based model

In this scenario, user-applications subscribe their interests with a message broker, and they express their willingness of receiving information published by specific data stream sources. When a subscriber is registered for specific publishers, it just has to wait the required information, according the push-based model. In a push-based model the massage broker plays the active role, it processes information from publishers, and it sends them to the registered subscribers. Inside the message broker subscriptions are stored as Continuous Queries, which will process all coming data from the selected data stream sources.

Since Continuous Queries have to execute for a period of time over an unlimited stream, it is really important to select the data that have to be considered in the execution. For this reason, it has been defined a mechanism to extract a finite number of tuples from an infinite data stream. This mechanism is based on the concept of Window that we will present in the next subsection.

1.2.3 The concept of window

Working with an unlimited collection of data, it is important to find a way to specify a finite set of elements to process. In order to do this, it is used *the concept of Window* that, since data streams cannot be fully archived, is usually applied over coming data and it represents a buffer that selects a sample of elements over the stream [17,11].

The basic configuration parameters of a Window are: the *range* and the *step*. The range of a window represents its size and the step represents the frequency of its repetitions on a stream [17].

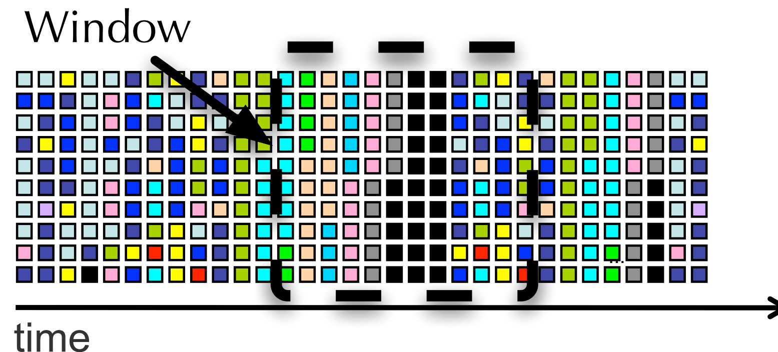


Figure 1.6 Logical example of Window on Data Stream **[11]**

As we have said, the window can be used into Continuous Queries, in order to identify a specific portion of the stream. An example of CQ may be: *calculate the average of cars passing through a specific tollgate with a step of an hour in a range of a day*. In this case a window considers the cars that are passed through the tollgate during an entire day and it calculates the incrementally average of cars every hour. Moreover the query is registered into the data stream processing system and it is executed every day, during a specific period of time.

1.2.4 Streaming Linked Data

Even if data streams are not usually part of the Web, they are becoming more often available on-line. There are several Internet sites that provide a way to distribute and share real-time information about every kind of topics. In this dissertation we take as example three Internet sites that can be interesting to develop a "smarter" city, and that represent a sample of existing technologies for describing a microenvironment of traffic monitor, a macro environment of a social network, and a global forecasting service:

- *Waze (http://world.waze.com/)*: it is a traffic monitoring site that can be able to provide a way to share traffic information between users (citizens) of a city, in order to calculate the better solution path to reach the designed point, based on users notifications;

- *Twitter (http://twitter.com/)*: it is an example of online social network service, whose aim is to realize a social network among people, to share their interests or activities;
- *AccuWeather (http://www.accuweather.com/)*: it is a web site that provides weather forecasting services worldwide.

Since data streams are starting to become part of Internet sites, they have to be subjected to the same semantic revolution of the Web. For this reason, *Streaming Linked Data (SLD)* have been defined as part of the Web of Linked Data [12]. The same as of Linked Data, SLD are collections of linked information that can be navigated by machines but, in this case, information has a dynamic nature.

Since SLD are machine-understandable information, it is possible to use a Rules-based Reasoner in order to extract the semantic meaning of the information. However SLD are extremely dynamic and the standard reasoning approach is no longer valid. In this scenario, the concept of *Stream Reasoning* realises a logical reasoning over real-time data streams, by exploiting the notion of Window [11]. In this way, the set of Entailment Rules used by the reasoner is applied over the Streaming Linked Data contained into the Window.

Working on stream, reasoning techniques are even more useful then working in a static environment, where the knowledge base is fully known in advance. If we consider for example a general static KB, one could actually avoid using reasoning techniques. Indeed it is possible to realise a manual reasoning and expressing a complex query with all the information that one may need. However, in real-time environments this cannot be done because necessary information may be not arrived yet, and one cannot know before which data will arrive.

1.2.5 Difference between Knowledge and Data

Even though we are talking about real-time environments, it is necessary to specify that not all data are real-time information, but they are also, for example, static descriptions of environments and sources.

Moreover, since data streams can be expressed as Streaming Linked Data they can also be described by ontologies, in the same way as Linked Data. Ontology represents the structure (*schema*) of information and for this reason, it can be

considered static as well. Both these types of data can be unified in a single term. In this dissertation we will differentiate between [18]:

- *Knowledge*: it is composed by schema information and static elements that, because of their nature, do not appear into data streams;
- *Data*: they are the actual stream information, those data that rapidly change over a period of time.

However this does not mean that *Knowledge* information is always static, but it means that is static during an *observation period*, the period in which the examined environment is subjected of a query.

To give a practical example of this division, one can consider a real-time environment of a smart city. In this case, names of streets, landmarks, kinds of events, etc. change very slowly, and they can be considered static during the observation period. On the other hand, since the number of cars that go through a traffic detector changes very fast, it represents real-time information [18].

At this point of the dissertation, we have presented all necessary basic concepts, which will be affected by our work.

2 BACKGROUND

This chapter presents a technical background to completely understand this work. It is based on the concepts that we have presented before, but here we will introduce more technical contents.

Moreover in the sections below, we will introduce some existing approaches to deal with problems of working on Streaming Linked Data that have been inspiring for our work (Sections 2.4, 2.4.2).

2.1 SEMANTIC WEB TECHNOLOGIES

As we have said, the World Wide Web was originally built for human consumption; the Semantic Web changes this assumption, adding semantic and machine-understandable meaning to Web information.

The Semantic vision of the Web is based on Linked Data, a structured organization of static information, which makes possible to bind concepts spread in different databases [4]. The Semantic Web provides multiple different technologies in order to build the Web of Linked Data, however in this chapter we will present a subset of them, those ones that are interesting for our work. Semantic Web technologies present in this chapter are:

- *Resource Description Framework (RDF)*: it is used to express Linked Data, and their semantic expression;
- *RDF Schema (RDFS)*: it is used to define a vocabulary of terms in order to express the semantic meaning of web information;
- *RDF Schema Entailment Rules*: they are a set of rules used by a Rule-based Reasoner, in order to extract the semantic meaning of Linked Data;
- *SPARQL Protocol and RDF Query Language*: it is the language used to query Linked Data.

All these technologies have been designed to work in a static environment, so they have to be extended in order to be able to describe and process real-time data. Such extensions are the topic of a later section (Section 2.3).

2.1.1 Resource Description Framework

Resource Description Framework (RDF) is a simple language description model to express Linked Data information. RDF is a W3C specification used to provide interoperability between applications that exchange machine-understandable information on the Web [3].

RDF is based on three different types of data [19]:

- *Uniform Resource Identifier (URI) references*: they are expressed as absolute URIs with optional fragment identifier. An example of URI is *http://www.w3.org /2000/01/rdf-schema#Class*. Since URIs can share common prefixes, the Qualified Name (QName) notation is often used for brevity and readability. QName notation defines a prefix element, for example: "`rdfs:`" expressed by a name and followed by colon punctuation. A prefix can be bound to an URI namespace that is the radix of the URI. In this case the URI namespace is *http://www.w3.org/200-0/01/rdf-schema#* (without Class). Thus the entire URI can be obtained by substitution of the notation "`rdfs:Class`". It is also possible to specify an empty string as a prefix name, and bound it to an URI namespace. If one considers the example of before, the QName notation will look like: "`:Class`".
- *Literal*: it is a number or a simple string of characters such as "`University of Bologna`" along with an optional data-type specification, useful to distinguish for example integers from strings. Literal elements represent concrete values and they are always expressed in quote notation (the value is contained into two quotes);
- *Blank Nodes*: they represent concepts that are either not known or not specified. They can define "anonymous" resources that are not identified by URIs. Since there is not a URI reference specified, the prefix name is expressed by a special notation: "`_:`". A Blank Node cannot be used to globally identify a resource, but its scope is local to the specific RDF document.

By exploiting the presented type of data, in the next subsection we will present the *RDF data model*.

2.1.1.1 RDF data model

The *RDF data model* is a way of representing RDF expressions without concern over syntax components. It is a conceptual modelling approach and it is based on the idea of making *statement* about data (in this case, web data) [3]. Since *RDF statements* are expressed in the form *subject-predicate-object,* they are called *triples*, and represent the smallest chunks of knowledge in an RDF document.

Subject predicate and object are the three parts of a RDF statement and they are expressed by [3]:

- *Resources*: everything described by RDF expressions and identified by URI or Blank Nodes is called resource. They can be subjects and objects of RDF statements;
- *Properties*: it is a specific aspect, characteristic, attribute, or relation used to describe a resource; Properties, like resources, are always identified by URI and they are predicates of RDF statements.

The object of a RDF statement can also be a RDF literal element, which represents a concrete value. Since the RDF data model is used to represent Linked Data, every single RDF statement can be represented as a graph, where subjects and objects are nodes and predicates are directed edges between them (Figure 2.1).

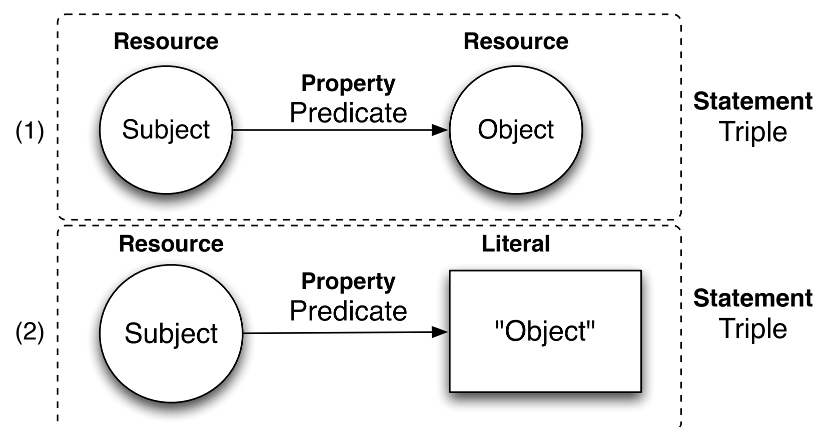


Figure 2.1 Logical representation of RDF statements

A practical example of RDF dataset is shown below. In this case, it has been used the QName notation with an empty string as a prefix name.

Considering the RDF triple, "`:Tom :drive :car`" (Line 1 of Table 2.1), in this case "`:Tom`" is the subject, "`:drive`" is the predicate, and "`:car`" is the object, but as one can see, "`:car`" can be also the subject of statements (Line 4, 5 of Table 2.1). The

RDF statement of the dataset describes that: *a person "*`:Tom`*" is 32 years old, he plays "*`:guitar`*", and he drives a "*`:car`*" that is black and that has 4 wheels.*

	RDF Dataset
1.	`:Tom :drives :car`
2.	`:Tom :plays :guitar`
3.	`:Tom :age "32"`
4.	`:car :color "black"`
5.	`:car :hasWheels "4"`

Table 2.1 Example of RDF dataset **[20]**

The same dataset can also be represented as a directed graph as shown in the figure below (Figure 2.2).

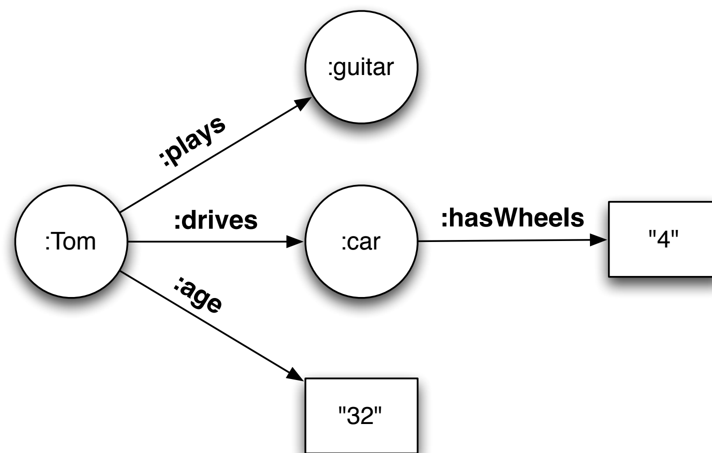


Figure 2.2 Example of RDF dataset as a directed graph **[20]**

The Resource Description Framework defines a simple model to describe Linked Data that can express the relationships among resources. Starting from this model in the next subsection we will present different serialization formats that have been defined to express RDF data.

2.1.1.2 Serialization formats

There are different kinds of serialization formats that from the RDF data model can produce a description document of a specific dataset.

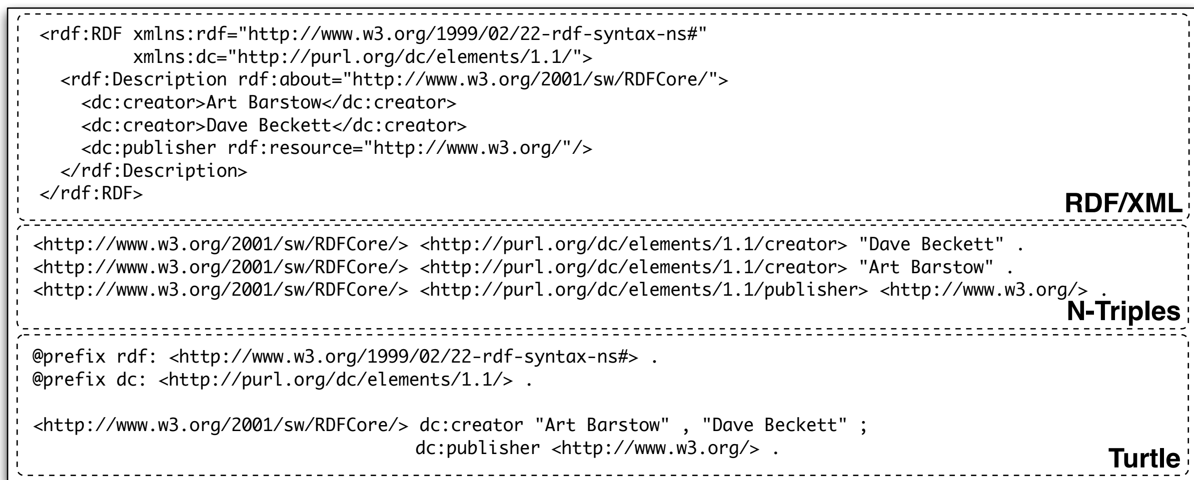


Figure 2.3 Representative set of RDF serialization formats

We present the three mains formats that are used to serialise RDF data model in RDF document (Figure 2.3):

- *RDF/Extensible Mark-up Language (RDF/XML)*: this format is also called simply RDF because it was introduced with the RDF specifications themself. In this case the RDF graph is encoded by nesting XML tags. Either XML tags or literals represent nodes of the RDF graph [21]. This format is the most verbose and the less easy to read, as one can see in the example above (Figure 2.3);
- *N-Triples*: it is a line-based plain-text language, where each line contains exactly one triple consisting of a sequence of subject, predicate, object, and eventually a data-type specification [22]. This format is frequently used, because it is simple to automatically parse.
- *Terse RDF Triple Language (Turtle)*: it is a line-based plain-text language that represents a super-set of N-Triples (Figure 2.3). Turtle can express multiple RDF triples in a single line and its specification improves the compactness and the readability of the data. Turtle has three mains shortcuts [23]:
    a. *comma separate notation*: it fuses together RDF triples that have same subject and predicate, shared by these triples;
    b. *semicolon separate notation*: it fuse together RDF triples that have same subject, shared by these triples;

c. *qualified name notation*: it applies the constructor "`@prefix`" that, as one can see in the example above, bounds the prefix name to the specific URI namespace.

As we said RDF is just a language to describe Liked Data, therefore RDF itself does not provide any mechanism to describe relationships between resources. This task is performed by *RDF vocabulary description language* that we will present in the next section.

2.1.2 RDF vocabulary description language (RDF Schema)

RDF vocabulary description language (RDF Schema or RDFS) is a semantic extension of RDF introduced to describe ontologies [24]. RDFS is able to define concepts and relevant relationships and model specific domains, defining vocabulary of terms expressed by RDF language [25]. RDFS is based on a set of default classes and properties that can specify other classes and other properties and so on, reaching the complete definition of a specific vocabulary of terms. The vocabulary of terms defines all possible concepts and all possible relationships between such concepts, to define any structure information to be used to describe data.

An important characteristic of RDFS is its *property-centric* approach [24], to easily define properties in terms of classes that they may be connected; instead of defining classes in terms of properties that may have, as usually happens in standard program languages. The benefit of this approach is the possibility to extend the description of existing resources without need of redefining the original description, according to the architectural principles of extendibility of the Web [24]. For example, it is possible to define a new property "`:author`" that has domain of "`:document`" and range of "`:person`". While in object-oriented languages, the property author is usually defined only as an attribute of a class document of type person. In that case, if we want to define another property of a document we need to change the class definition. On the other hand, the RDF approach has been made to be extensible, and a third person can easily define additional properties with domain of "`:document`" and range of "`:person`".

The next subsection presents the default classes and properties that are used by RDFS in order to define the semantic meaning of data.

2.1.2.1 RDFS classes and properties

The core RDFS vocabulary is defined by *http://www.w3.org/2000/01 /rdf-schema#* usually expressed in QName notation by the prefix "`rdfs:`", while all the other elements are identified by *http://www.w3.org/1999/02 /22-rdf-syntax-ns#* usually expressed by the prefix "`rdf:`". Moreover, note that in all examples listed below we will use the Turtle serialization format [24].

RDFS provides several default classes and properties, to describe the semantic information of data. However in this subsection we do not deal with all of them, but just with those more relevant to describe the structure of the information, and with those ones that are interesting to explain the set of entailment rules that are used to describe the semantic information that RDFS describes. We will present the entire basic set of RDFS classes, plus other classes useful to express the concept of collection of data. After that we will present the mains properties, used to describe attributes of data.

The basic set of RDFS classes is composed by [24]:

- *"`rdfs:Resources`"*: it is the class that contains all the possible elements, since everything that can be described by RDF is a resource. This means that every RDF resources are instances of the class "`rdfs:Resources`";
- *"`rdfs:Class`"*: it is the class of resources that are RDF classes. Note that every "`rdfs:Resources`" is instance of "`rdfs:Class`", which is also instance of itself;
- *"`rdf:Property`"*: it is the class of resources that are RDF properties. In the same way as all the classes are instance of "`rdfs:Class`", all the property are instances of "`rdfs:Property`",
- *"`rdfs:Datatyp`"*: it is the class of all the possible data types of the RDF model. Since every data type is described by concrete values (URI) all of them are instances of "`rdfs:Literal`".
- *"`rdfs:Literal`"*: it is the class of all concrete values. They are expressed by literal values such as strings and integers. As we have said, literal values can have an optional data type specification. In this case the typed literals are instances of the class "`rdfs:Datatype`". This is not true for literals without data type specifications;
- *"`rdf:XMLLiteral`"*: it is the class of XML literal values. It is an instance of "`rdfs:Datatype`" and a subclass of "`rdfs:Literal`".

As we have said, RDFS provides also classes used to describe the collection concept:

- *“rdfs:ContainerMembershipProperty”*: it is a class whose instances are all properties, and they are used to state that a RDF resource is member of a collection;
- *“rdfs:member”*: it is the super-class of all instances of “rdfs:ContainerMembershipProperty”.

Now that we have presented all the classes, we can describe all the properties that can be exploit to specify relationships between data. In this case as before, we will not analyse all the possible properties, but just those ones that are more relevant and also present in the Entailment rules. In the following paragraphs, we will present [24]:

- *“rdf:type”*;
- *“rdfs:subClassOf” and “rdf:subPropertyOf”*;
- *“rdf:range”*
- *“rdf:domain”*;

“rdf:type” is a central RDFS property that is used to describe the membership of a resource in a class [19]. “rdf:type” defines the relation: *instance of,* between a subject (instance) and an object (class) [24]. For example the RDF statement below states that: *K is instance of C.*

K rdf:type *C*.

“rdfs:subClassOf” and “rdf:subPropertyOf” are RDFS properties used to state respectively that a class is sub-class of another class, and that a property is sub-property of another property. In this way all the instances of a sub-class (sub-property) are also instances of another sub-class (sub-property) [24]. In the example below a generic class (property) *C1* is defined as sub-class (sub-property) of *C2*, consequently, a generic *I* instance of *C1*, it is also instance of *C2*.

C1 rdfs:subClassOf *C2*.
C1 rdfs:subPropertyOf *C2*.
I rdf:type *C1*

“rdf:range” is used to state that values of a property (objects of RDF statements) are instances of one or more classes, specified as values (RDF objects) of “rdf:range” [24]. In the example below, it is defined a generic RDF statement S P O (subject, property, and object) where the property *P* has range *C* (class), and where for this reason *O*, object of *P*, is also instance of *C*.

P rdf:range *C*.
S P O.

"rdf:domain" is used to state that every resource that has a given property (subjects of RDF statements) is an instance of one or more classes, specified as values (objects) of "rdf:domain" [24]. In the example below, it is defined a generic RDF statement S P O (subject, property, and object) where the property *P* has domain *C* (class), and where for this reason *S*, subject of *P*, is instance of *C*.

P rdf:domain *C*.
S P O.

Now that we have presented all the "tools" to describe data, we use these items to provide an example of a vocabulary of terms for a simple RDF dataset (Table 2.2).

	RDFS Description
1.	:person rdf:type rdfs:Class.
2.	:student rdf:type rdfs:Class.
3.	:document rdf:type rdfs:Class.
4.	:student rdfs:subClassOf :person.
5.	:bachelorDegreeFrom rdf:type rdfs:Property.
6.	:masterDegreeFrom rdf:subPropertyOf :bachelorDegreeFrom.
7.	:author rdfs:range :document.
8.	:author rdfs:domain :person.
	RDF Dataset
9.	:Tom rdf:type :student.
10.	:Tom :masterDegreeFrom "University of Bologna".
11.	:Tom :author <http://.../TomThesis>

Table 2.2 Example of RDF and RDFS dataset

The above example expresses that: *Tom is a student master degree of Bologna University author of the thesis* <http://.../TomThesis> (Lines 9, 10, and 11 of Table 2.2). As one can see, every single element of this sentence is further defined using RDFS classes and properties. *Person*, *Student*, and *Document* are defined as instances of "rdfs:Class" (Lines 1, 2, and 3 of Table 2.2) and *Student* class is further defined as sub-class of *Person* (Line 4). In line 5 is specified that ":bachelorDegreeFrom" is an instance of "rdfs:Property", and super-property

of "`:masterDegreeFrom`". The last property defined (Lines 7 and 8) is "`:author`", with range *Document* and domain *Person*.

There are several RDF Schemas already developed and ready to be used, which compose the knowledge base of the Web. In order to have an example one can see the *Friend of a Friend Project (FOAF)* that is interesting to describe the social relation between people and their activities [26]. It provides a set of classes and properties identified by the URI namespace *http://xmlns.com/foaf/0.1/*, and that is usually referred by prefix "`foaf:`".

2.1.2.2 Web Ontology Language (OWL)

Even if in our work we deal just with RDFS data, for completeness this subsection defines the *Web Ontology Language (OWL)*, another important "piece" necessary to build the Semantic Web vision.

The *Web Ontology Language* is a W3C recommendation, and the same as RDFS, it is used to describe the semantic meaning of data. OLW is based on RDF and RDFS, but it presents more facilities for expressing semantic information than RDFS. For this reason, OWL goes beyond RDFS in ability of representing machine-understandable contents on the Web [27]. By extending the property of RDFS, OWL is able to define richer vocabulary of terms and consequentially more expressive data descriptions [27].

2.1.2.3 Ontology Hijacking

Existing ontologies are designed to work by following the extendibility principles of the Web, so that everyone can extend an existing ontology, and produce specific *legacy concepts*: by re-defining the existing ones with another ontologies. In this directive, it becomes really important to specify a way in order to produce a discipline for the extension of ontology, by following the advice against *"Ontology Hijacking"* [28]. The Ontology Hijacking is either the re-definition or extension of a definition of a legacy concept in terms of classes or properties by other ontologies, which modify the inference results on such legacy concepts [28]. This means that one could re-define an existing ontology by introducing new and not authorized classes and properties. The re-definition can vastly increases the amount of RDF

statements that are inferred during the materialization phase and it can potentially damage the inference on data contributed by other parties [28].

The example below shows the problem of re-definition of the class "`onto1:Person`" belonging to the Ontology1 with a class "`onto2:Person2`" belonging to Ontology2.

```
onto1:Person rdfs:subClassOf onto2:Person2
```

In this way, because the definition of "`rdfs:subClassOf`" property, all instances of the class "`onto1:Person`" are also instances of the class "`onto2:Person2`". In this way the "`onto1:Person`" property would be hijacking, effecting the processing of all RDF statements that contains "`onto1:Person`".

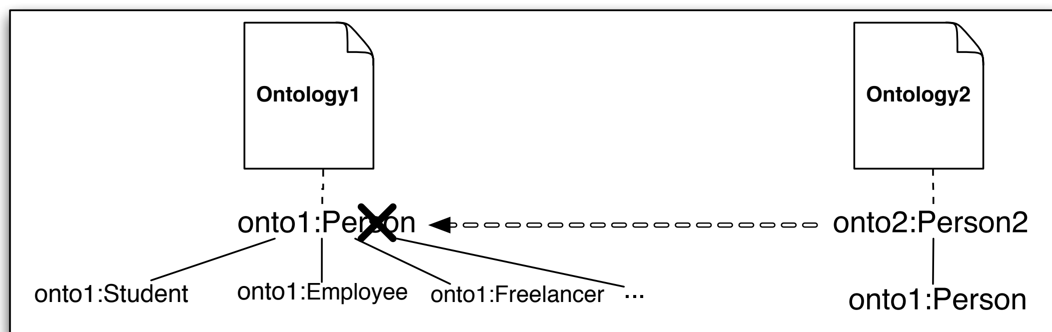


Figure 2.4 Example of *Ontology Hijacking*

A correct extensions of the "`onto1:Person`" class is shown in the example below.

```
onto2:Person2 rdfs:subClassOf onto1:Person
```

In this way there is no change into the existing knowledge base and RDF statements with "`onto2:Person2`" property can be added without inference problems.

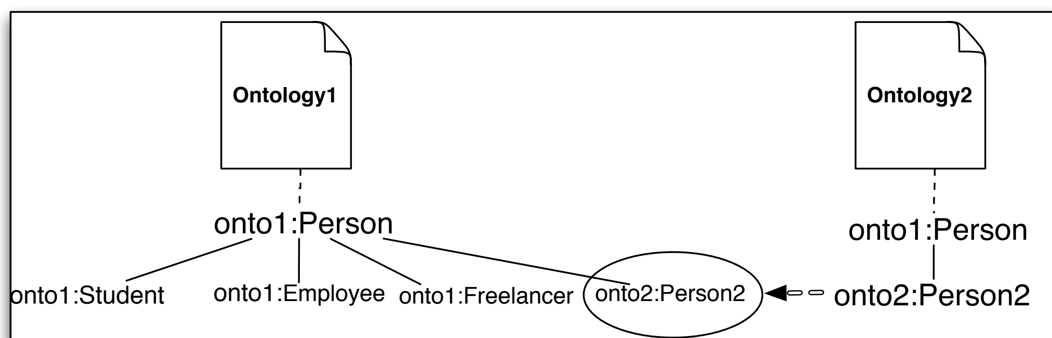


Figure 2.5 Example of correct ontology extension

2.1.3 RDF Schema Entailment Rules

Even if RDF Schema is able to express the semantic meaning of information, the defining of the structure and the relationships among data, it is not possible to extract this information without entailment rules. There are different sets of rules to deal with different types of ontology; more expressive ontologies (as OWL ontologies) required more extensive set of rules.

We have assumed to work with RDF Schema ontologies, and for this reason, we have exploited the *RDF Schema Entailment Rules (RDFS Rules).* RDFS Entailment Rules are a set of thirteen basic inference rules defined by W3C to calculate the complete materialization of RDFS ontologies [29].

Each RDFS Rule is expressed by using RDF description language and it is composed by: a RDF triple for the head (or conclusion) of the rule and one or more triples for the body (assumptions) of the rule, as shown below.

```
Triple1 & Triple2 ➔ Triple3
```

All thirteen RDFS Rules are listed in a table placed at the end of this section (Table 2.3). We will use the number of every line of the table to identify the associated rule. Into the next list, we will provide a formal definition of every single rule, in order to be able to further understand how they infer new knowledge. However after we will consider again this rules at the end of this section, by giving a more informal and simplified definition [29].

- *Rule 1*: it states that from every RDF triples (*S P O*) with a literal object, it is possible to derive a triple with a Blank Node (associated to the object) instance of the class Literal.
- *Rule 2*: it states that if a property *P* domain of V is also predicate of a generic RDF triples (*S P O*), then the subject *S* is an instance of V.
- *Rule 3*: it states that if a property *P* range of V is also predicate of a generic RDF triples (*S P O*), then the object *O* is an instance of V.
- *Rule 4a* and *4b*: they are used to state that every subject and object of RDF triples is instance of the class Resource.
- *Rule 5*: it states the transitive property of the sub-property predicate;
- *Rule 6*: it states that every property *P* (instance of the class Property) is the sub-property of itself.

- *Rule 7*: it states that all resources characterised by one a property P_2 are also characterised by a property P_1, if P_1 is a sub-property of P_2.
- *Rule 8*: it states that every class instance of RDF Class is also a sub-class of the class Resources.
- *Rule 9*: it states that if C_1 is a subclass of the class C_2, and I is an instance of C_1, then I is also instance of C_2.
- *Rule 10*: it states that every class C (instance of the class RDFS Class) is the sub-class of itself.
- *Rule 11*: it states the transitive property of the sub-class predicate.
- *Rule 12*: it states that every instance of class "`rdfs:ContainerMembersh-ipProperty`" is a property M, which is also a sub-property of the class Member.
- *Rule 13*: it states that every instance of the class Datatype is also a sub-class of the class Literal.

All these rules express the same concepts that we have explained in the previous section presenting the RDFS classes and properties. For this reason, some of them express also some basic concepts like: *every property is a sub-property of itself* (rule 6). Therefore, starting from the complete set of RDFS rules (Table 3.5), one can define a subset without loss generality [30]. Indeed it is actually possible to avoid considering part of the rules that present a single assumption and that cannot be used for further non-trivial derivations. This means that these rules are not necessary in the activation chain generated by a reasoner to evaluate Linked Data materializations, because from their conclusions cannot be used to derive other useful conclusions [30]. The set of rules that is not strictly necessary are rules 1, 4a, 4b, 6, 8, and 10 (Table 2.3). Therefore, now the defined subset of rules is composed by rules 2, 3, 5, 7, 9, 11, 12, and 13 (Table 2.3) that are those rules that we will deeply analyse in Chapter 5 and that we will use in order to develop our Rule-based Reasoner.

	Complete set of RDFS Entailment Rules *Assumptions*	➔	*Conclusions*
1.	S P O *O is a plain literal*	➔	_:n rdf:type rdfs:Literal *_:n identifies a blank node allocated to O*
2.	P rdfs:domain V & S P O	➔	S rdf:type V
3.	P rdfs:range V & S P O	➔	O rdf:type V
4a.	S P O	➔	S rdf:type rdfs:Resource
4b.	S P O	➔	O rdf:type rdfs:Resource
5.	P_1 rdfs:subPropertyOf P_2 & P_2 rdfs:subPropertyOf P_3	➔	P_1 rdfs:subPropertyOf P_3
6.	P rdf:type rdf:Property	➔	P rdfs:subPropertyOf P
7.	P_1 rdfs:subPropertyOf P_2 & S P_1 O	➔	S P_2 O
8.	S rdf:type rdfs:Class	➔	S rdfs:subClassOf rdfs:Resource
9.	C_1 rdfs:subClassOf C_2 & S rdf:type C_1	➔	S rdf:type C_2
10.	S rdf:type rdfs:Class	➔	S rdfs:subClassOf S
11.	C_1 rdfs:subClassOf C_2 & C_2 rdfs:subClassOf C_3	➔	C_1 rdfs:subClassOf C_3
12.	M rdf:type rdfs:ContainerMembers- hipProperty	➔	M rdfs:subPropertyOf rdfs:member
13.	D rdf:type rdfs:DataType	➔	D rdfs:subClassOf rdfs:Literal

Table 2.3 RDF Schema Entailment Rules **[29]**

To exploit this set of rules is only a way to extract the semantic information, and for this reason, they are indispensable in the Semantic Web vision. However, these rules do not have to be used manually, but a Rules-based Reasoner exploits them in order to extract the semantic meaning of the data materialising all needed conclusions from the rules.

2.1.4 SPARQL Protocol and RDF Query Language

SPARQL is a recursive acronym for *SPARQL Protocol and RDF Query Language* [31]. It is a declarative query language, W3C recommendation, and one of the key components of the Semantic Web. SPARQL works on static Linked Data in order to retrieve information from knowledge bases distributed on the Web. It represents a radically change in traditional declarative query languages, *Structured Query Language (SQL)* based.

In the next subsections we present a briefly comparison between SPARQL and SQL focus on their main different characteristics, then we will deeply study the "anatomy" of a SPARQL query and their mains components.

2.1.4.1 SPARQL and Structured Query Language

SPARQL and SQL are both declarative languages used by user-applications to express request to the query engine. However they provide also other additional capabilities so to be exploited by different types of systems.

SQL is used on relational databases and it is able to find data from tabular representations [32]. The SQL query selects the correct row of a table and retrieves the correct values, contained into specified columns. In the example below (Figure 2.6) the query, *select the name of the graduated student whose ID is 5,* looks for the column identified by "`idStudent`" and then the correct row to satisfy the constraint ("`idStudent = 5`"). Hence it retrieves the "`Name`" of the graduated student: "`Tom`".

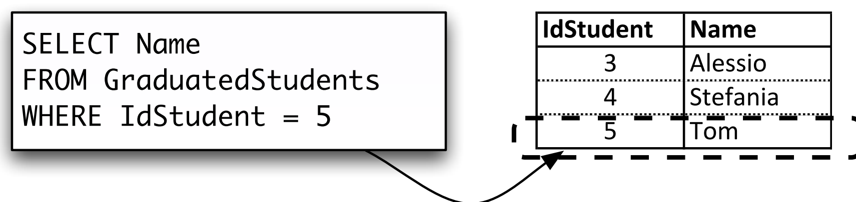


Figure 2.6 Example of SQL query behaviour

Note that, if we work in a distributed environment as the Web environment, where there are different and heterogeneous systems interconnected, the same data that are stored in table can be stored differently in another one. There is not a specific standard about how to define information in a tabular representation. In different databases the same data may be expressed in different tables with different column names and so on. In this scenario the integration between databases can be really complex [33].

On the other hand, SPARQL works with Linked Data technology, where the information is stored as knowledge based and representable as graphs. For this reason, the traditional approach is no longer valid. To deal with this characteristic a SPARQL engine uses the *pattern-matching* paradigm to answer the queries. The pattern matching is able to select branches of the graph that are interested by the

query. Moreover, it is able to extract the results and bind the selected values with the query variables.

In the example below, one can see a SPARQL query executing on a RDF graph: the dataset of information is the same as the example above (Figure 2.6), even if in this case data are organized as a graph. The SPARQL query looks for a graduated student associated with id 5, then it selects the graph branch identified by "`:Stude3`" and it retrieves the student name.

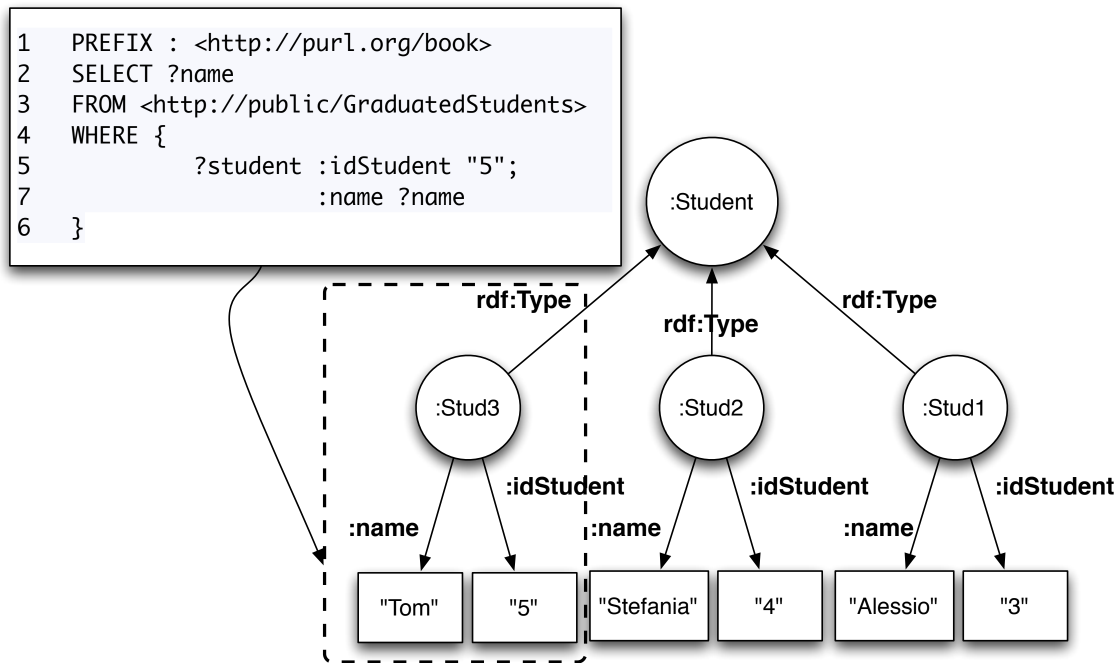


Figure 2.7 Example of SPARQL query behaviour

SPARQL is designed to merge different sources of data and querying information on the Web, so there are no problems for execution in distributed source systems. Contrary to SQL, SPARQL and RDF's foundation are using a standardized data model that is made to simplify the exploration and embrace and extend existing schemas in an open data vision [33].

2.1.4.2 SPARQL query structure

This subsection presents the different form of a query, its structure, and their mains components.

There are different forms of SPARQL queries that can be used to perform different actions or return results in different forms:

- *select query*: it is used to identify values of variables that are selected by the query. It is similar to the standard SQL query form, and it is able to return all, or a subset of variables bound by in the pattern matching phase;
- *construct query*: it is used to identify values of variable that are parts of RDF statements. For this reason, a *construct query* is used to return an RDF graph constructed by substituting variables in a set of triples;
- *ask query*: it is used to identify if a specific pattern matching is found or not in a RDF graph and it returning a Boolean value.
- *describe query*: it is used to identify the branches of an RDF graph that describes the resources found with the pattern matching, it does not return variable values, but the ontology description that is interested by the pattern matching;

Even if there are four different forms of queries, all of them, have to deal with the pattern matching on a graph, even if they present the results in different ways. Furthermore, with the exemption of the *select query* forms, all the other ones are extremely rare [34] and there are few useful use cases. For all these reasons, in our work we decide to implement just the *select query* form, and consequently in this subsection we present just this form of queries.

It is possible to identify different mains parts that compose the structure of a SPARQL query [31]:

- *the declaration of prefix shortcuts*: used for abbreviating URIs and for applying the QName notation. They are identified by the keyword *PREFIX*;
- *the query result clause*: it can be used for identifying which information have to be returned from the query, and for specifying variables that have to be selected. It is identified by the keyword *SELECT* plus variable names;
- *the dataset definition*: used for specifying which RDF graph(s) are being queried. It is also possible to specify more than one graph from different sources. The dataset definition is identified by the keyword *FROM* or *FROM NAMED* plus the URI reference to specify the RDF graph;
- *the query pattern*: it is the most important part of a query that expresses all constraints to identify a specific pattern matching on the underlying RDF graph. The pattern is expressed in RDF Turtle format and is also called *graph pattern*,

because as RDF triples they can be represented as a graph. Its aim is to match specific part of the graph and bound query variables with the graph values. This part of the query is identified with the keyword *WHERE*. We will analysis with more details all possible graph patterns in the next subsection;

- *the query modifiers*: the query operators that rearrange the query results, like [31]:
    a. the *slice* operator: it applies an upper and or lower threshold on the number of solutions returned by the query. The upper bound may be specified by using the *LIMIT* keyword. The lower bound the *OFFSET* keyword.
    b. the *order* operator: it applies an ascending or descending order to the solution set. The operator is identified by the *ORDER BY* keyword.
    c. the *aggregate* operator: it applies expressions over some parts of the result set. By default the solution consists of a single group containing all results. This grouping clause may be specified using the *GROUP BY* keyword. Moreover aggregate functions allow operations such as counting (*COUNT*), numerical min/max/average (*MIN/MAX/AVG*) and so on, over grouped parts of the solution.

All these components of a SPARQL query structure can be found in an example of SPARQL query shown below (Figure 2.8).

```
1   PREFIX book: <http://purl.org/book>
2   SELECT ?title
3   FROM <http://public/library>
4   WHERE {
5            ?book book:title ?title .
6   }
7   ORDER BY ?title
```

Figure 2.8 A simple SPARQL query

The query of the example above selects all book titles of the library (*http://public/library*), ordering the query results by title. The SPARQL variables of the query are indicated by "*?*" or "*$*" (Figure 2.8, line 5) and they are added to the standard RDF statement format.

In this case the query pattern is really simple and formed by a single RDF triple, however is also possible to specify multiple types of query patterns, more complex ones can be formed by combining smaller patterns in various ways. In the next subsection we will present all possible graph pattern that one can find in a SPARQL query.

2.1.4.3 SPARQL graph patterns

All possible graph patterns are expressed by RDF turtle format. Even if in this case, all possible element of the RDF triple can be variables. As we said, the variables are defined with "*?*" or "*$*" keyword and they can be used instead of every element of the RDF statement, as in the example below.

```
?subject ?predicate ?object
```

All possible types of graph pattern that one can find in a SPARQL query are [31]:

- *Basic Graph Pattern (BGP);*
- *Group Graph Pattern;*
- *Optional Graph Pattern;*
- *Alternative Graph Pattern;*
- *Patterns on Named Graphs;*

Basic Graph Pattern (BGP) is the simplest Graph pattern it is formed by a set of RDF triples that have to be considered all together. In this way, if all triples find a match on the graph, it is possible to select specific branches of the RDF graph. After all branches are found, the variables are bound with RDF terms, producing a pattern solution. A pattern solution of a BGP is a sub-graph of the respective RDF graph (Figure 2.8).

Group Graph Pattern is a set of one or more graph patterns delimited by braces. In this case, all graph patterns have to match in order to select the results, as one can see in the above figure. The BGP is the simplest example of this kind of pattern.

Optional Graph Pattern is set of optional triple patterns identified by the *OPTIONAL* keyword. It defines an optional graph pattern that can or cannot match. If the pattern matches, the specified variables are bound and added to the pattern solution. If the pattern does not match it leaves the solution unchanged without adding any additional bindings.

Alternative Graph Pattern is a set of two or more possible triple patterns, where all possible pattern solutions of each single graph pattern are found. This pattern is

specified by the *UNION* keyword and the solution of every single pattern are unified together, in order to have a single solution.

Patterns on Named Graphs is a set of triple pattern that are matched against specific named RDF graph (graph identify by URI). Moreover in this case, the URI that identified the graph may be part of the solution. A Pattern on Named Graph is specified by using the *GRAPH* keyword.

Other two fundamentals elements of a graph pattern are the filter clause (identified by the *FILTER* keyword) and the sub-query. The filter clause is used to reduce the number of results that have matched, based on some specific constraints. The Sub-query is a query nested in another query, inserted into the graph pattern of the parent query, and it is used to nest the results of a query within another query [31].

In order to have a practical example of how a graph pattern works, we can consider the follow RDF dataset (Table 2.4). It defines the knowledge base of the library identified by the URI *http://public/library*. The library contains two different books "`:book1`" (line 3) and "`:book2`" (line 4).

	RDF Dataset - Library
1.	@prefix book: <http://purl.org./book/>
2.	@prefix : <http://purl.org./example/>
3.	:book1 book:title "SPARQL Tutorial" .
4.	:book1 book:title "Advance SPARQL" .

Table 2.4 RDF dataset – Library

Consider again the SPARQL query of Figure 2.8, the query selects all book titles of the library *http://public/library* that match the BGP. If we execute this query on the RDF dataset in the table above, the query finds all matches of the BGP (all RDF triples that contains the predicate "`book:title`") and then it binds the variables "`?book`" and "`?title`". The variable "`?book`" is bound with "`:book1`" and "`:book2`". The variable "`?title`" is bound with "`SPARQL tutorial`" and "`Advance SPARQL`". Now that the variables are bound the query result clause select the values of the variable "`?title`" that are expected in the results set.

```
1   PREFIX book: <http://purl.org/book>
2   PREFIX ex: <http://purl.org/example>
3   SELECT ?title
4   FROM <http://public/library>
5   WHERE {
6            ?book book:title ?title .
7            :Book1 book:title ?title.
8   }
```

Figure 2.9 SPARQL query with multiple triple patterns

If the SPARQL query presents graph pattern with multiple triples (Figure 2.9), every triple pattern is matched on a RDF graph. However the result set of the query contains just the value that satisfies both of triples. In this case, by executing the query on the same RDF dataset of before (Table 2.4) the result set will contain just the title of "`:book1`": "`SPARQL tutorial`", because even if the first triple (Line 6) will match both books, the second triples (Line 7) will select the "`book:title`" of "`:book1`". Therefore the final results will contain the elements that match both triples.

The query engine executes the evaluation of a query, finding the answer to the SPARQL query. There are different possible implementations of a SPARQL query engines, one of the most famous is *ARQ*, which is the query engine of *Jena framework*, it provides an open sources framework to develop Semantic Web applications Java based. Another existing system to query RDF dataset is *Sesame* that is similar to Jena and that can be used to implement semantic web applications that are working with SPARQL and Linked Data.

2.2 DATA STREAM PROCESSING

Data Stream Processing is a new paradigm of data processing typically of real-time environments. In this context, the high volume of dynamic data and the complex real-time analysis have forced to change the management of information. Applications require to process large volumes of continuously arriving data with high throughput and low latency. Furthermore, since information is an unlimited data stream they cannot be stored, but they have to be consumed on the fly. For these reason, as said in the previous chapter, the traditional concept of database is no

longer valid. In this scenario we introduce the *Data Stream Management Systems (DSMSs)* to substitute traditional *Database Management Systems (DBMS)* [35].
A *Database Management System* is a software system that uses a standard method of cataloguing, retrieving, and running queries over data. It works on static information contained in traditional relational databases. In order to retrieve this information a DBMS express query with SQL based languages that are used to realise specific request and obtain single response, following the pull-based access data model [36].
Data Stream Management Systems can be seen as extensions of *DBMS* that also supports data streams: in fact, a DSMS can work on both transient and persistent information executing both traditional and continuous queries. Continuous queries are register into the DSMS and automatically executed during a period of time. Such queries produce continuous results, following the pub/sub push-based data model.
Note that in this scenario, if a DSMS is not fast enough to process the stream information those data that are not read are lost [36]. The next section will analyse the architectural model of a DSMS, focus on the component that a DSMS provides.

2.2.1 Data Stream Management System architectural model

The aim of this subsection is to present the architectural model of DSMS and the components that are necessary to deal with data streams.

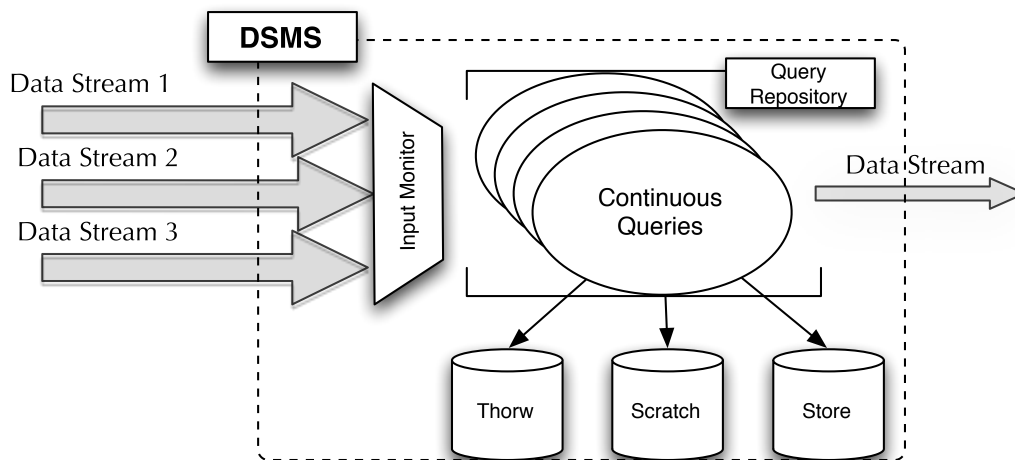


Figure 2.10 DSMS architectural model

A Data Stream Management System is modelled as a set of continuous queries, registered in a *query repository* and accessed by the query engine to process

information. Moreover, a DSMS is equipped with an *input monitor,* whose task is dropping excessive tuples from the stream, in order to not saturate the system. This component is important because the rate of the stream is unpredictable, and it could exceed the computational capacity of the system (CPU cycles and memory size) that could produce substantial loss of data (Figure 2.10). Moreover, in order to produce continuous query evaluations a DSMS produce four possible outputs (Figure 2.10) [37]:

- the *stream*: it is formed by all tuples that are into the stream of the answer. Since these tuples are themselves a stream, the results are produced one time and they are never changed, following the time-dependent nature if data in a stream that after a while have to be discarded;
- the *store*: it is a storage of that part of the answer that may be changed or removed with futures update. The *stream* and the *store* together define the current complete answer of a query;
- the *scratch*: it represents the working memory of the system, it is a "place" where is possible to store intermediate results or useful elements to compute the answer;
- the *throw*: it represents the recycle bin of the system, it is used to throw away all non necessary tuples.

Since continuous queries are the central part in DSMS architecture we will present them with more details in the next subsection.

2.2.2 Continuous Query

Continuous query represents a high level specification to define *transforming rules* that will execute on data streams. Transforming rules are the execution plans of the queries that define a flow of operator, which are charged to "transform" coming tuples on the data streams following the rules specified with the query. These transforming rules are implemented by *streaming applications* that define a set of operators, which can be linked together in order to transform data streams [37,38]. Each operator can take flows of discrete items (tuples) as inputs and produces new tuples as output, which can be forwarded, exploiting stream connections to others operators or directly sent them out, as one can see in Figure 2.11.

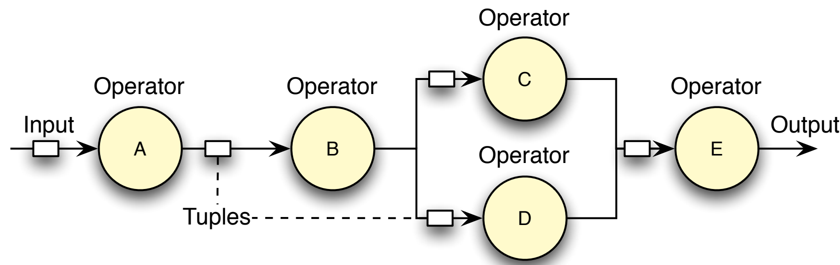


Figure 2.11 Logical representation of a streaming application

Existing DSMS systems offer different types of operators, to execute different queries, specifying one or more operations that process the input flows. Those operators perform filtering, joining, and aggregating to the received data. Since they are working on data stream they need to exploit the concept of window, however not everyone needs a window on the stream. It is possible to divide operators in two classes:

- *stateless operators*: operators that do not need to use the window over the stream and whose results do not depend on other elements in the stream. An example is the Filter operator, which can define constraints to reduce the number of the output tuples. If the constraints are verified the tuples can pass through the operator, otherwise they are filtered;
- *stateful operators:* operators that need to exploit the window over the stream. In this case the window specifies a range of elements, where can be applied the specific operator functionality. An example is the Join operator that is used to correlate tuples from two streams, based on selected elements of the stream that are contained into the range of the window configuration.

Streaming applications can be specified in two different ways, they can be either *compiled* or *user-defined*, completely defined by users. In the first case the system itself compiles the specified execution plan (transformation rules) of the continuous query; in the second case, the system allows defining the exact flow of operators that are responsible to transform the information. This type of streaming applications are usually better performing then the compiled ones, because they can be configured specifically to work on determinate domains.

There are a wide range of DSMS offers in both research and commercial domains. In the next subsections we consider an overview of two existing systems.

2.2.3 Existing Data Stream Management Systems

In this subsection we present two existing DSMSs to manage data streams, from research and commercial domains, respectively The *Stanford stream data manager (STREAM)* and *IBM infoSphere Streams*. These systems are a representative example of DSMSs that exploit the two different way to define continuous queries: compiled and user-defined.

2.2.3.1 Stanford stream data manager

Stanford stream data manager (STREAM) is a prototype developed at Stanford University that implements a system to manage data flows. It was one of the earlier stream processing systems proposed in the academic community [39].

```
1   SELECT *
2   FROM S1 [Rows 100], S2 [Range 2 Minutes]
3   WHERE {
4             S1.A = S2.A
5        And S1.A > 10
6   }
```

Figure 2.12 Example of Continuous Query Language query **[39]**

STREAM exploits a *Continuous Query Language CQL* to define continuous queries that are compiled in streaming applications (Figure 2.12). CQL is a declarative variant of the SQL to work on data stream. For this reasons, it provides the concept of window definition on the stream that is essential to select a range of elements that are interested by query operators. An example of CQL query is shown in Figure 2.12, where is possible to see that the query is really similar to SQL, even if it presents a modified dataset specification (*FROM*), with window configurations applied on the inputs. The query in the example joins the element *A* of the last 100 tuples of S_1, if *A* is greater then 10, with the tuples of S_2 that have arrived in previous 2 minutes.

When the CQL query is defined, it is compiled and transformed into a streaming application. STREAM system provides two types of working domain: the *Stream*, that we have already presented, and the *Relation*. This one represent a working domain out of the stream, where is possible to apply standard relational operators (join, aggregate, union, and etc.).

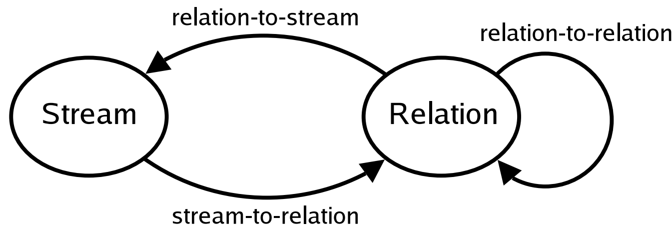


Figure 2.13 Model of STREAM continuous queries **[39]**

For this reason in order to pass from one domain to another and process the stream information, STREAM has defined three classes of operators [39]:

- *relation-to-relation*: they are directly derived from operator of standard database system. They are the core of STREAM streaming application, those ones that actually process the information;
- *stream-to-relation*: they implement the notion of window. They are responsible to select elements from coming streams and to store this information into the relational working domain, where the processing takes place;
- *relation-to-stream*: they define how processed tuples can become part of a new information flow. These operators transform the results, from the relational working domain to the stream, adding the new tuple to the data flow.

As we have already said, in STREAM a declarative language to compiles streaming applications that are responsible to process the stream. However in this way, the generated applications cannot be configured with all necessary details, because they are obtained from a high level language. It is not possible, for example, to explicit specify the connection between operators or express custom functionalities. This limitation is no longer valid in IBM InfoSphere Stream that we will present in the next subsection.

2.2.3.2 IBM InfoSphere Streams

IBM System S Streams, aka *InfoSphere Streams*, is a commercial stream-computing platform that provides computing infrastructure for processing large volumes of possibly high rate data streams with broader spectrum of input data modalities [40].
Since it is a widely used business infrastructure, it provides several additional features for processing data streams, as the possibility to deploy streaming applications on clusters, to improve the execution performance, and so manage a

larger amount of data. InfoSphere Streams divides the streaming applications are into basic operators called *Processing Elements (PEs)*, which are connected to each other in order to compose a complex flow of operators. These streaming application are specified with *Streams Processing Language (SPL)*, a declarative language that is able to realise a distributed data flow composition. It descends from the *IBM Streams Processing Language (SPADE)* and it embeds all traditional relational database functionalities in native operators that are used to filter, join, aggregate, and etc. stream information [40].

SPL is able to specify user-defined stream applications and the high configuration level allows realising a complete definition of the entire stream application, without need of understanding the lower-level stream-specific operations. This happens because SPL as a declarative language provide a set of operators that can be easily combined together. Moreover, SPL allows users to write their own PEs, custom operators that can exploit a full featured programming language, and an Integrated Development Environment (IDE). In other words, users can write virtually every kind of function, by explicitly deciding all possible behaviours. Moreover, as said, InfoSphere Streams is also designed to be deployed on a cluster, and run continuous queries on a collection of cooperating multiple machines. In this way, it can be used to support processing of large-scale data scenarios [41].

InfoSphere Streams, like other systems that provide user-defined continuous queries, is able to give the level of configuration that one needed in order to realise specific streaming applications. For these reason we decide to exploit this system as data stream processing system to deal with all possible heterogeneous data streams coming from real-time environments. We exploit the complete configuration of the operators flow in order to provide a query engine and a Rules-based reasoner that are able to process such data streams. For these reasons we will further analyses InfoSphere Streams in more details in section (Section 2.5) giving an overview of the system and SPL functionalities.

2.3 STREAMING LINKED DATA PROCESSING

As we have sated several times that, data streams are appearing more often on the Web as real-time sources. In that scenario, Streaming Linked Data are the new technology of the Semantic Web to work on real-time information. In the same way

as Linked Data, RDF statements can be used to express Streaming Linked Data, even if RDF itself does not provide any mechanism to express time-dependency of data. For this reason one needs to extend RDF: typically the *RDF stream*. RDF stream has been defined to work on stream information and, for this reason, it can be defined as data streams: ordered sequences of couples of tuples and timestamps [13]. However in this case, each couples being made of an RDF triple (`< sub pred obj >`) and a timestamp (τ_i)

< < sub1 pred1 obj1 > >, τ_1>
< < sub2 pred2 obj2 > >, τ_2>
< < sub3 pred3 obj3 > >, τ_3>
…

In the same way as in data stream, timestamps associated to the RDF tuples have to be considered as sequences of monotonically non-decreasing values [13]: that exploits the concept of timestamp, to realise RDF stream able to define time-dependents RDF triples, which can be discarded after a while, based on the window configuration.

However, as we have said, not all information is real-time data, therefore not all data need to be expressed as Streaming Linked Data. Considering again the difference between Knowledge and Data (Subsection 1.2.5). Linked Data (Knowledge) and Streaming Linked Data (Data) have to be used together in order to produce a proper static and dynamic description of real-time environments.

We have already seen example of Linked Data (Table 2.2), therefore we now present a practical example of Streaming Linked Data: they represent a sequence of students that are subscribing courses at the university ("`:subscribe`").

< < :Tom :subscribe :curse1 > >, τ_1>
< < :Ste :subscribe :curse2 > >, τ_2>
< < :Jon :subscribe :curse1 > >, τ_3>
…

By exploiting Streaming Linked Data, it is possible to produce a new type of knowledge base that is represented by RDF streams. This new KB is extremely dynamic and based on data streams. In this scenario, the simplest SPARQL declarative language has to be extended in order to deal with the new dynamic nature of SLD information.

In the same way as DSMS continuous queries exploit the concept of window to work on data streams, SPARQL extensions need to define a window to deal with Streaming Linked Data. The queries specified with a SPARQL extension are types

of continuous queries that provide the same pattern matching paradigm of SPARQL but, as any other continuous queries, they exploit the facilities provided by a DSMS for processing data streams: continuous query processing and pub/sub push data model.

In the section below we will focus on all possible window configurations that can be defined in order to working on data streams and Streaming Linked Data.

2.3.1 The window configurations

We have already defined the concept of window (Subsection 1.2.3), however in this section, we are interesting in presenting a deeply analysis of all kinds of window configurations that can be defined to work on Streaming Linked Data, and, in general, on every type of data stream.

The concept of window, used to specify a finite subset of items from the streams, is based on two fundamentals parameters: range and step, which can be combined in order to produce different configurations.

It is possible to identify three different kinds of window ranges [17]:

- *time-based*: it defines a range based on a period of time (Figure 2.14) that cannot be expressed in the past. Since data streams have to be consumed on the fly, the range usually starts from now plus a specific period of time. For example a time-based window can detect all cars that are passed through a tollgate during the last ten minutes.
- *tuple-based*: it defines a range based on the number of tuples that have passed through the stream without time condition. For example can be useful to detect the first ten cars that have passed through a tollgate;
- *marker-based* or *punctuation-based*: they specify a range based on special messages (markers) that are flowing into the stream with data and can be used to identify the final element of window. It is further possible to define a marker that is based on specifics elements of the stream and, in this way, it is possible to use this information to identify the last element of the window. For example, it is possible to detect all cars that have passed through a tollgate before a bus.

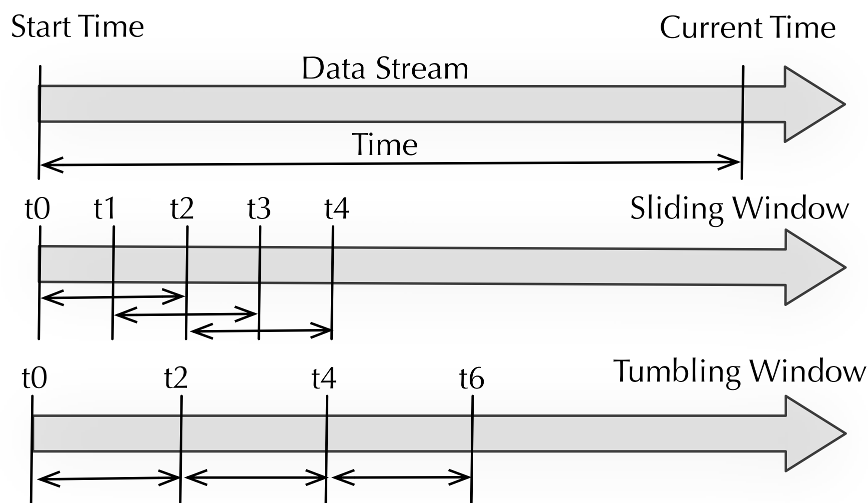


Figure 2.14 Sliding and tumbling time-based window

Furthermore, it is also possible to classify the window based on two different step configurations (Figure 2.14) [17]:

- *sliding window*: where the step of the window is different to the value of the range of the same window, and in this way, the different windows can be overlapped or completely distinct. For example, for a ten seconds window with a five seconds step, the overlap if five seconds for every window;
- *tumbling window*: where the step of the window is the same of the value of the range of the same window, and for this reason, the overlap of every window is zero. Example of this configuration is a window with ten seconds step and 10 seconds range.

In Figure 2.14 one can see an example time-based sliding and tumbling window, however similar representations can be done for tuples-based and marker-based window.

There are several SPARQL extensions that are used to deal with Streaming Linked Data and Linked Data and we will analyse some of them in next chapter of this dissertation (see Chapter 3).

2.3.2 Stream Reasoning

Since *Streaming Linked Data (SLD)* are machine-understandable information, it is possible to apply the set of entailment rules of a Rules-based Reasoner, in order to

calculate the complete materialization of data, and in this way, extract the semantic meaning of information. However SLD are extremely dynamic because like data streams they are given in real-time, therefore the standard reasoning approach is no longer valid. In this scenario, have to be defined the concept of *Stream Reasoning.*

Stream Reasoning realises a logical reasoning over Linked Data Stream, exploiting continuous data processing and the notion of window [11]. In this way the materialization is calculate continuously and in a finite selection of tuples of the stream (Figure 2.15).

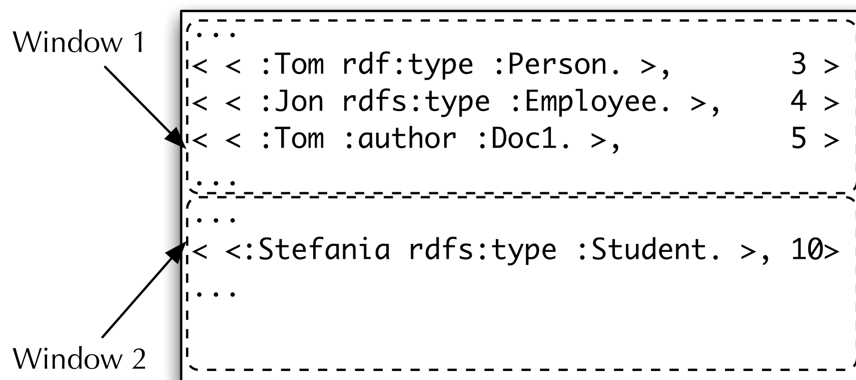


Figure 2.15 Window on RDF stream

Because Streaming Linked Data are also data streams, a Rule-based Reasoner for SLD is realised as a streaming application, which manages the stream information and that calculates the necessary materialization exploiting the inference rules.

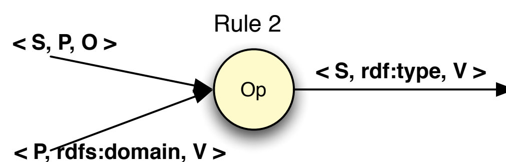


Figure 2.16 Stream reasoning – RDFS rule 2

In this scenario, every inference rules can be represented as an operator of a streaming application, as one can see in the figure above (Figure 2.16), where the RDFS rule 2 (Table 2.3) is represented as a generic operator with two input and a single output stream: the two inputs represent its assumptions, and the single output its conclusion. In this way, every rule can find its assumptions in the Streaming Linked Data and produce new conclusion that become part of the stream. However, since the conclusions of some rules are assumptions for other rules, they have to be

fed back as inputs into the reasoner, and in this way it is possible to calculate the complete materialization of elements selected by the window.

Existing approach of stream reasoning are Rule-based reasoner that works in forward reasoning mode [42], by exploiting a parallel methods to compute the complete materialization of a large amount of RDF stream triples. However this kind of approach is really expensive, because the elements into the window are continuously changing and for this reason the complete materialization have to be calculated in every new window.

For this reason, we have decided to exploit a backward reasoning approach, in order to reduce the number of triples into the window that need to be processed to prove the goal, in this way, we can reason over the limited necessary set of RDF stream triples. We will present our backward reasoning approach in Chapter 5 of this dissertation.

2.4 EXISTING STREAMING LINKED DATA PROCESSORS

In this sections we will focus on two existing systems to process Streaming Linked Data, which provide two opposite approaches:

- *Continuous SPARQL (C-SPARQL)* [43];
- Continuous Query Evaluation over Linked Streams *(CQELS)* [44].

C-SAPRQL is one of the most cited systems to query Streaming Linked Data, and it provides a complete SPARQL extension definition to work on streams, by exploiting existing systems in order to process Linked Data and Linked Data Streams [43,35].

CQELS*,* as C-SPARQL, provides a well-defined SPARQL extension definition, but with a complete different approach: it provides a native system to work on Linked Data and Streaming Linked Data. CQELS provides a performant query execution by exploiting native DSMS operators in order to process liked information [44]. We present also CQELS, because our system implementation to process Linked Data and Streaming Linked Data is based on the same idea of the CQELS approach: by exploiting native operator to process Linked Data and Streaming Linked Data information.

Note that, the sections below (both sections 2.4.1, 2.4.2) present the two systems without focus on their SPARQL extension definitions, to be analysed in the next chapter (Chapter 3).

2.4.1 Querying Streaming Linked Data with DSMS and SPARQL

Continuous SPARQL (C-SPARQL) has been defined to work on real-time environments and query large amount of dynamic information, flowing into the stream [43]. It has been defined as an extension of SPARQL, used to produce continuous queries over Streaming Linked Data, however we will present the language features in the next chapter (Chapter 3).

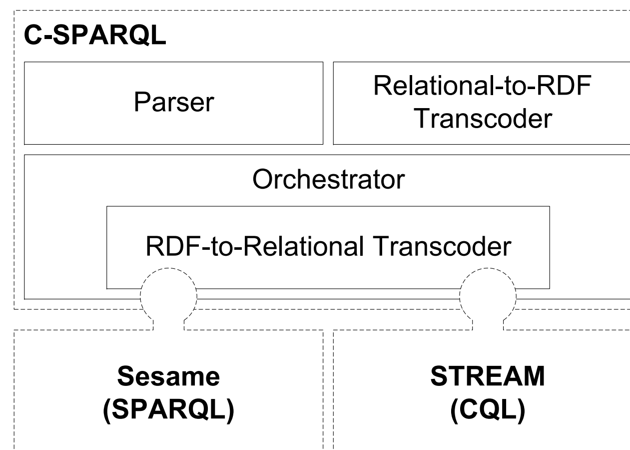


Figure 2.17 C-SPARQL architecture overview

The C-SPARQL execution framework is based on the assumption that information can be divided in Knowledge and Data, by following the idea presented in the previous chapter (Section 1.2.5). For this reason, every C-SPARQL query can be divided in a static and dynamic part, where the static part works on the Knowledge and the dynamic part works on the Data. This split is really important in order to be able to produce a query evaluation reusing existing systems. A Data Stream Manage System can deal with the Data, and a standard SPARQL query engine can deal with the Knowledge [35].

The C-SPARQL execution framework is based on a plug-in platform independent architecture. Even if, this particular implementation uses STREAM, as DSMS, and Sesame, as SPARQL query engine, to run the dynamic and static parts of C-

SPARQL queries. As one can see in the architecture overview, it is possible to identify two main components (Figure 2.17) [35]:

- the *parser*: it parses and transforms a C-SPARQL query in such way that can the orchestrator handle them;
- the *orchestrator*: the central component of the framework that split the static and dynamic conditions of the query, and sends them to the respective query executors.

C-SPARQL is not able to directly process the information as RDF stream, but it needs to transform such data. The stream information (the Data) is selected with the current window and transformed in to static data that are merged together with the standard RDF datasets (the Knowledge). In this way, the query is executed at the same time on both types of data [35]. The dynamic part of the C-SPARQL query is responsible for selecting the dynamic elements from the stream, while the static part calculates the results of the current query on both static and dynamic data (managed in static). However since C-SPARQL demands the execution to other two existing system, both of static and dynamic parts of the query need to be re-written: in CQL to execute on STREAM, and in standard SPARQL language to execute on Sesame.

The power of C-SPARQL approach is to re-use existing systems, which have been tested and developed over the last decade, in order to realise an execution environment to process Streaming Linked Data. However, the query translation, the data transformation, and the delegate execution present important limitations, because they adopt a "*black box*" approach [44], where every execution is further delegates to other systems such STREAM and Sesame. This kind of approach, together with the query translation, does neither allow full control over the execution plan, nor over the implementation of the query engine. For this reasons, the possibilities for query optimisations are very limited [44].

2.4.2 A native and adaptive query processor for Streaming Linked Data

Continuous Query Evaluation over Linked Streams (CQELS) is a native and adaptive query processor that unifies query processing over Streaming Linked Data and Linked Data [44]. CQELS provides a flexible query execution framework, where the query dynamically adapts their execution based on input data. In this case, native operators can directly process Streaming Linked Data and Linked Data [44].

If C-SPARQL uses a “black box” approach, the CQELS exploits the opposite one, a “*white box*” approach [44], where the native operators to process information, can be completely configured and optimised to improve the query execution. CQELS is build on the top of *System S Streams*, IBM stream process system (Subsection 2.2.3.2) that can provide a complete specification of the data flow operators [45].

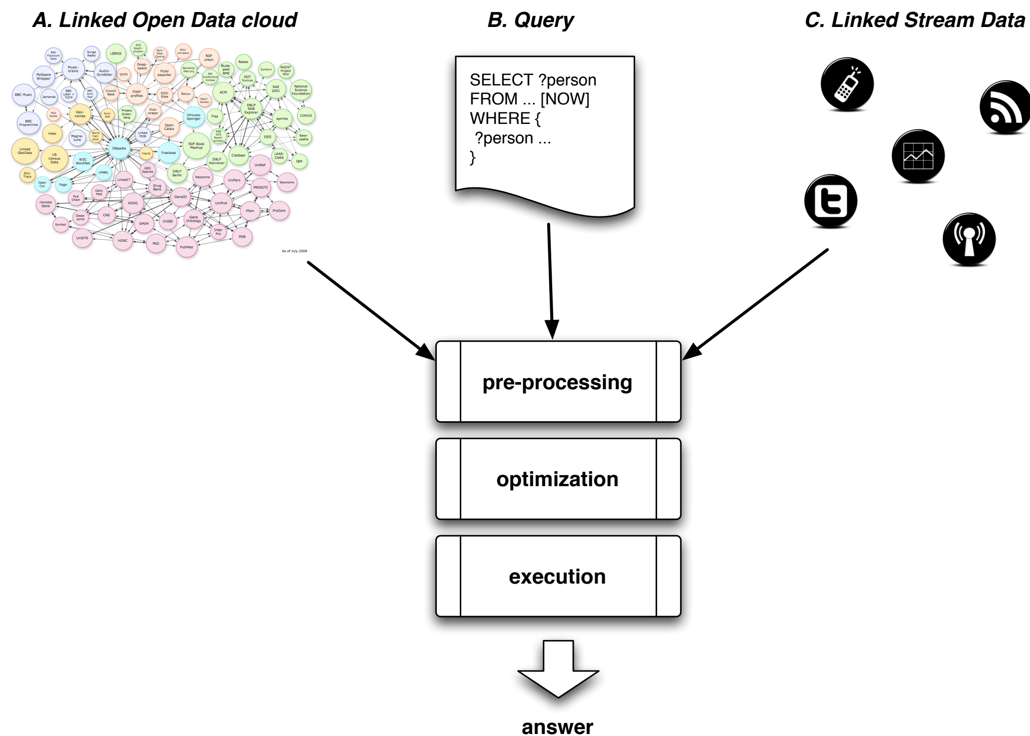


Figure 2.18 CQELS processing model

As one can see in the Figure 2.18, CQELS takes as its inputs, Linked Data, Streaming Linked Data, and a Query specification. These inputs are inserted into the processing model, which is based on three different phases (Figure 2.18) [45]:

- *pre-processing*: responsible for the optimisation of the data representation and storage. It transforms the information in a compacted format, making easer loading larger amounts of data into the memory system, during the query execution. Furthermore, it is also used for storing intermediate query results;
- *optimization*: responsible to select the least expensive execution plan. It chooses between a set of pre-computed plans, that calculates selectivity values using a cost-based strategy;
- *execution*: responsible to run the query and return the results.

In order to process Linked Data and Streaming Linked Data, CQELS requires dedicated operators that can be identified in three groups [44]:

- *Window Operators*: they take snapshots of the input stream and they filter the valid elements based on some configured conditions;
- *Relational Operators*: they operate on finite data, to compute standard relational operators;
- *Streaming Operators*: they convert the final results back into the stream, in a similar way as the relational-to-stream, operator of CQL.

The CQELS approach realised a powerful execution framework that is focused on the optimization of the query execution.

2.5 INFOSPHERE STREAMS AND STREAMS PROCESSING LANGUAGE

As we have said, *IBM InfoSphere Streams*, alias *System S Streams* [41], is a commercial high-performance computing platform that can provides a rapidly ingest and analysis of information as theirs arrives from thousands of real-time sources. Like other DSMS, InfoSphere Streams is designed to perform continuous queries, exploiting operator data flows, which in this case are named *Processing Elements (PEs)*. The *Streams Processing Language (SPL)* is the language used to specify streaming application, this language descends from *IBM Streams Processing Language (SPADE)* that embeds all traditional relational database functionalities in native operators that are used to filter, join, aggregate, and etc. stream information [40]. SPL is designed to give the user control over the most *performance-critical* aspects of the streaming application: it is possible to directly control the graph topology, the operator connections, and the data representations. Therefore it exposes enough information to enable optimisations, however at the same time, it is high-level language by keeping the low-level details of operator implementations blind to the users.

SPL is based on two kinds of operators [41]:

- *primitive operators*: they are the native operators of the system and are written in a native language (C++ or Java). In this way it is taking advantage of the performance and productivity of traditional languages for straight-line code;

- *composite operators*: they represent a reusable sub-graph of streaming applications. They are written in SPL and composed by *primitive operators*.

The available primitive operators are contained in SPL toolkits where they are subdivided by their specific application domains. The always-available toolkit is the *SPL Standard Toolkit Reference* that contains all standard operators to realise streaming applications. All these primitive operators can be grouped in [41]:

- *relational operators*: they perform the standard operation of filtering, joining, and aggregation that one can find in the traditional database system;
- *adapter operators*: they are responsible to manage the different inputs sources available and the different possible output modalities;
- *utility operators*: they realise general operations to manage data streams.

Furthermore InfoSphere Streams allows users to write new operators. It is actually possible to define new primitive operators C++ or Java based, but also to realise custom operators exploiting the utility operator *Custom*. In this second way the operator behaviour is completely specified by SPL that can apply peculiars functionalities on the incoming tuples [41].

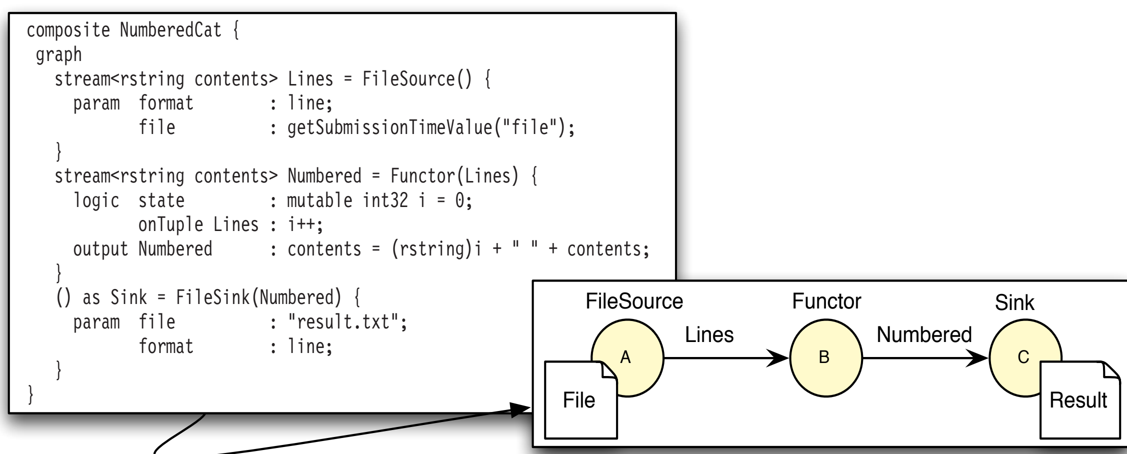


Figure 2.19 SPL Code and graphical representation **[46]**

The example above gives the idea of how an SPL streaming application looks like, and how it can be represented (Figure 2.19). In this case the streaming application is reading lines of a file, and it is writing the same lines to another file numbering every line. However we will present SPL operator details in Chapter 4 of this dissertation.

2.6 ARQ ENGINE OF JENA FRAMEWORK

As we have said, ARQ is the query engine of Jena that supports SPARQL and that can be used to query different RDF graphs, providing an optimised query evaluation. ARQ produces a simplified representation of a SPARQL query by using the *SPARQL Syntax Expressions (SPARQL S-expressions* or *SSE)* representation. SSE is a language used to print the low-level operators of SPARQL, defined by the *SPARQL Algebra*. The *SPARQL algebra* defines a set of relational operators that can be used to translate the query in simplified form that gives all well-defined execution steps of a SPARQL query. We will study the details of this translation from a SPARQL query to SSE query in Chapter 4 of this dissertation [47].

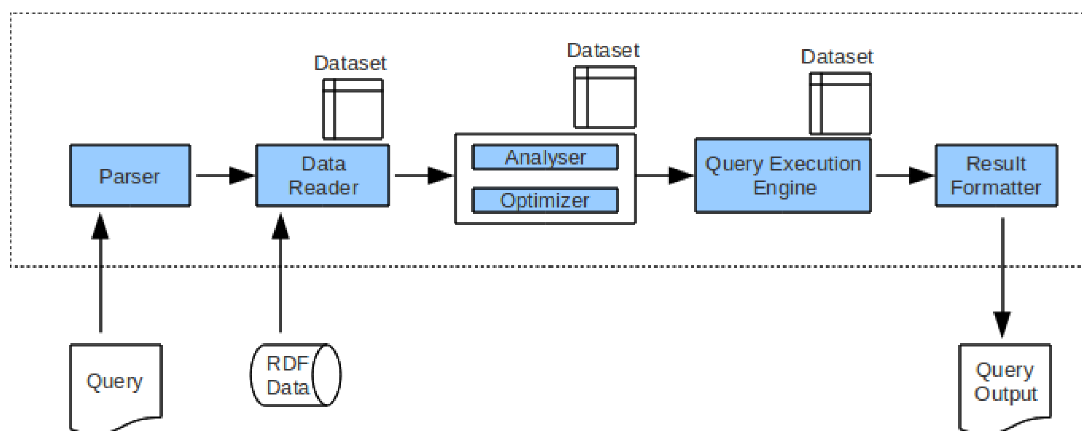


Figure 2.20 ARQ architecture overview **[20]**

The logic ARQ architecture (Figure 2.20) can be divided in five mains components [47,20]:

- the *Parser*: it takes as input a query string, a query expressed in SPARQL language, and it provides an internal query representation: a *query object*;
- the *Dataset Reader*: it reads and saves in memory the input RDF datasets before the query execution;
- the *Analyser*: it converts the query object in SPARQL Algebra expressions, so it defines a specific query evaluation plan;
- the *Optimiser*: concurrently to the analyser it is also performed a query optimization. In this phase are re-organised and re-ordered the operators to obtain a minimised intermediate result set size.

- the *Engine*: it is responsible to execute the query evaluation and to provide the query output.

In our work, we do not use ARQ as a query engine, but only two components of the ARQ architecture. We are interested only in the query Parser and the query Analyser in order to obtains an intermediate representation of the query string to further transform the query object in a streaming application.

With ARQ, our introduction is completed and we now have all necessary elements to continue through the next chapters of this dissertation, where we will provide a complete description of our work.

3 THE NEW SPARQL EXTENSION DEFINITION

We have presented the expressivity power of Linked Data and Streaming Linked Data in the previous chapters: at this moment, real-time information is still offered in vary heterogeneous formats, not necessary similar to RDF or RDF streams. In some real-time environments there are different data serialization formats for exchange real-time information, which in an integrated vision could be considered and could be queried in the same way to produce a unique and useful service able to describe the entire environment [48].

For example, in a smart city use case, real-time information comes from different and heterogeneous sources like: buses, pedestrian, pollution, noise sensors, and so on, providing heterogeneous serialization formats. For our work, we have exploited real-time information coming from *Dublin City,* where the bus positions are given in *Service Interface for Real Time Information (SIRI)* [49] (XML based)*,* common protocol of exchange real-time information about public transport services and vehicles [49], and noise, pollution levels, and pedestrian count are given in *Comma Separated Value (CSV)* format. This information can be queried together with RDF stream information coming, for example, from traffic monitor Internet sites (like *Waze*), weather forecast services (like *AccuWeather*), or with static RDF information of the position of amenities of Dublin City.

Since, there are no SPARQL extension able to query such heterogeneous formats simultaneously, without transforming the data in RDF, we want to provide a new SPARQL extension that is able to provide this behaviour. We have decided to realise a new SPARQL extension starting from two existing extensions: C-SPARQL and CQELS, and providing features to manage streams given in CSV format.

We have decided to natively support CSV format because it is the most common one, it is widely supported, by consumer, business, and scientific applications, and it can be easily obtains from any other serialization formats.

In the first two sections of this chapter we will present the two SPARQL extensions: C-SPARQL and CQELS that we have already analysed in the previous chapter,

however in this case, we will focus on their language features that can be useful to realise a new SPARQL extension for heterogeneous data stream.

3.1 QCELS

Continuous Query Evaluation over Linked Streams is a native and adaptive query-processing engine for querying Linked Data and Streaming Linked Data. We have already presented this approach in previous chapters. But now we are interested in analyse the different language features proposed by QCELS. In Figure 3.1 one can see the definitions of the CQELS extension [44].

GraphPatternNotTriples → GroupOrUnionGraphPattern |OptionalGraphPattern | MinusGraphPattern | GraphGraphPattern | **StreamGraphPattern** | ServiceGraph-Pattern | Filter | Bind

StreamGraphPattern → '*STREAM [*' Window '*]*' VarOrIRIref '*{*'TriplesTemplate'*}*'

Window → Range | Triple | '*NOW*' | '*ALL*'

Range → '*RANGE*' Duration ('*SLIDE*' Duration)?

Triple → '*TRIPLES*' INTEGER

Duration → (INTEGER '*d*' | '*h*' | '*m*' | '*s*' | '*ms*' | '*ns*')+

Figure 3.1 CQELS SPARQL extension definition **[44]**

CQELS is bound to SPARQL grammar with a new element added into the graph pattern definitions of SPARQL (*GraphPatternNotTriples* Figure 3.1), that contains all possible patterns that can be defined and that we have presented in Subsection 2.1.4.3. QCELS add its stream graph pattern (*StreamGraphPattern*), specifically built to work data streams, into the possible graph pattern specifications. The *StreamGraphPattern* is defined by the *STREAM* keyword and it contains, a standard triple pattern *(TripleTemplate)*, the window configuration, and the URI specification of the RDF stream source interested *(VarOrIRIRef)* [44].

As one can see, there are two window configurations available: *time-based (Range)* and *tuple-based* (*Triple*). The first one can be defined with range *(RANGE* keyword*)* and step *(SLIDE* keyword*)* that are expressed with an integer plus a time unit

(d,h,m,s,ms, and ns). On the other hand, the *TRIPLES* keyword plus the number of triples define the tuple-based window configuration [44].

As one can see in the figure below, there are two examples of *StreamGraphPattern* configurations: the first one (Lines 1 and 2) defines a triple pattern with a triple-based window; the second one (Lines 4 and 5) defines a triple pattern with a time-based window [44].

```
1  STREAM <http://.../rfid> [TRIPLES 1]
2         {?sub :pred ?obj}
3
4  STREAM <http://.../rfid> [RANGE 30s]
5         {?sub :pred ?obj}
```

Figure 3.2 CQELS stream graph pattern examples

CQELS is based on SPARQL language and it can use all SPARQL clauses to create a complex query. In this way CQELS can unify Linked Data and Streaming Linked Data, as one can see in the example of Figure 3.3, where the static input (identified by *FROM NAMED*) is merge together with two input streams. The two streams are time-based (30 seconds) and tuple-based (1 triples) window configurations that can be applied selectively on stream graph patterns. Through these skills the query is able to count the number of co-authors appearing in nearby locations in the last 30 seconds, grouped by location. As one can see, the detection of the co-authors, since is a stream information, is realised by the stream graph pattern, while the general data about the co-authors are selected from standard graph pattern, since they are statics.

```
1  SELECT ?loc2 ?locName count(distinct ?coAuth) as ?noCoAuths
2  FROM NAMED <http://.../floorplan/>
3  WHERE {
4          STREAM <http://.../rfid> [TRIPLES 1] {?auth lv:detectedAt ?loc1}
5          STREAM <http://.../rfid> [RANGE 30s] {?coAuth lv:detectedAt ?loc2}
6          GRAPH <http://.../floorplan/> {
7              ?loc2 lv:name ?locName.
8              ?loc2 lv:connected ?loc1 }
9          { ?paper dc:creator ?auth.
10            ?paper dc:creator ?coAuth.
11            ?auth foaf:name ''$Name$'' }
12  FILTER (?auth != ?coAuth)}
13  GROUP BY ?loc2 ?locName
```

Figure 3.3 Example of CQELS query **[44]**

By exploiting these capabilities, even if CQELS is able to realise complex query on Linked Data and Streaming Link Data, it cannot perform on other data formats, so it is not a suitable language to work in a heterogeneous environment as a smart city. If for example, we could have stream information from a traffic sensor and a second flow of Streaming Linked Data provided by a Web site of weather forecasting (*AccuWeather*), in this set, we might need to simultaneously query this information to be able to find a relationship between the weather and the traffic situations. However if the traffic sensor does not provide data in Streaming Linked Data format, we cannot answer to this request.

3.2 C-SPARQL

C-SPARQL is the same as CQELS, a SPARQL extension to query Streaming Linked Data and Linked Data that we have already presented in the previous section (Section 2.4.1). Here we will focus now on the C-SPARQL languages features [13].

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

Registration → (*'REGISTER STREAM'* | *'REGISTER QUERY'*) QueryName
[*'COMPUTED EVERY'* Number TimeUnit] 'AS' Query

Figure 3.4 C-SPARQL SPARQL extension definition **[13]**

Figure 4.4 shows can see the definition of each operator of the C-SPARQL extension. In this case, C-SPARQL does not have any new graph pattern, but it provides a new dataset definition in order to include data streams in the standard inputs of the SPARQL query. The clause is specified with *FromStrClause* and it is identified by the keywords: *FROM STREAM* or *FROM NAMED STREAM*, plus a specific URI that identifies the stream source (*StreamIRI*). Moreover, always in the

dataset specification, it also provides the window configuration, identified by the *RANGE* keyword of the specific stream [13].

As CQELS, C-SPARQL provides two type of window configuration: tuple-based, named *physical window (PhysicalWindow),* and time-based, named *logical window (LogicalWindow)*. The physical window is identified by the keyword *TRIPLES* plus the number of triples into the range, while the logical window is identified by a time value of the range, plus a time value for the step, that is given by the window overlap (*WindowOverlap*) that is identified by the *STEP* keyword [13]. The different time values are qualified by time units (*timeUnit*) that are used to specify days, hours, minutes, and so on. By exploiting these features, C-SPARQL like CQELS is able to realise query on Linked Data and Linked Data Stream, however it is also subjected to the same limitation: it cannot work on data format different from RDF.

C-SPARQL presents also the concepts of *Registration*. The Registration can be very useful in order to register the query in the DSMS and to specify the computation frequency of the entire query. In this way, the results of a previous execution have to be updated from the next execution. If we consider this skill in the set of a smart city, we can imagine hundreds of continuous queries performing specific monitoring services. In this set, the continuous queries could exploit the registration features, in order to be available when a citizen may need one of these services.

There are two types of registrations: *REGISTER QUERY*, used to register select or ask queries whose results are single value, and *REGISTER STREAM*, used with construct and description query whose results are RDF and RDF stream. In both cases the query can be registered with a specific query name (*QueryName*) and can be additionally specified *COMPUTED EVERY* keyword, which is used to declare the frequency at which the query should be computed [13].

```
1  REGISTER QUERY OpinionMakers COMPUTED EVERY 5m AS
2  SELECT ?opinionMaker
3  FROM STREAM <http://.../interact.trdf> [RANGE 30m STEP 5m]
4  WHERE { ?opinionMaker foaf:knows ?friend.
5          ?friend ?opinion ?document.
6          ?opinionMaker ?opinion ?document.
7          FILTER ( timestamp(?friend) > timestamp(?opinionMaker)
8                   && ?opinion != sd:accesses ) }
9   GROUP BY ( ?opinionMaker )
10  HAVING ( COUNT(DISTINCT ?friend) > 3 )
```

Figure 3.5 Example of C-SPARQL query **[13]**

Furthermore C-SPARQL provides a definition of the function *timestamp*, which is used in order to manage the RDF stream timestamps and bound it to the linked element of the graph pattern (Line 7 in Figure 3.5), in this way, it can be able to express the succession of events into the stream. Figure 3.5 shows an example of this functionality, where the query identifies all users (“`?friend`”) who are likely to influence the behaviour (“`?opinion`”) of other users, by matching interactions of the same kind (“`?opinionMarker`”) that occur on the same document (“`?document`”) after the first user has performed them. Therefore, by exploiting the timestamp function (“`timestamp(?opinioinMarker)`”), it is possible expressed the dependency in time necessary to find all users that have been influenced by other ones.

3.3 SPARQL FOR HETEROGENEOUS DATA STREAMS

Into the previous sections of this chapter we have defined two different SPARQL extensions, to produce query on Streaming Linked Data. However we need to define a new SPARQL extension to process heterogeneous data streams: we want to be able to query simultaneously Linked Data, Streaming Linked Data and data stream expressed in CSV.

We have decided to natively support CSV format because it is the most common one, it is widely supported both, by consumer, business, and scientific applications, and it can be easily obtained from any other serialization formats. Moreover in our use case, most information is given in CSV and in XML (following the *SIRI standard*) formats, which can be easily transformed in CSV.

As everyone knows, a CSV is a practical format that refers to collections of data expressed as plain text, divided in record per line, and further subdivided in fields (called columns) separated by delimiters (colon or tab). However we consider CSV data streams, where every line of the CSV can be represented as tuple of the stream that is divided in different attributes, as one can see in the example below.

$$< \text{Bus1,Latitude1,Longitude1,}\tau_1 >$$
$$< \text{Bus2,Latitude2,Longitude2,}\tau_2 >$$
$$< \text{Bus3,Latitude3,Longitude3,}\tau_3 >$$

…

As we have said, we want to produce a new SPARQL extension: to query both Streaming Linked Data and Linked Data, we based our extension on two already presented existing SPARQL extensions: C-SPARQL and CQELS. Mixing their advantages we have produced a further extension to manage CSV data stream.
In the subsections below, we previously identify the important language features we are interested (Subsection 3.3.1) and then we will provide our new SPARQL extension definition (Subsection 3.3.2).

3.3.1 Analysis and comparison

In order to work on heterogeneous data stream, we need to extend the existing SPARQL grammar [31], by exploiting two well-defined approaches: C-SPARQL (Figure 3.4) and CQELS (Figure 3.1), in order to have at list of guidelines to extend SPARQL. We identified a set of fundamental features that our language needs to expose, and analysed the two languages in order to provide the best way to express such features:

- the *window concept*;
- the *management of timing*;
- the *query registration*;
- *multiple input streams*;
- *CSV input streams*;
- *CSV graph patterns*.

The window concept represents a central extension of the SPARQL grammar in order to work on streams. Both C-SPARQL and CQELS present the window time-based and tuples-based specifications, even if they are expressed in different modals. C-SPARQL defines the window configuration into the dataset specification (FromStrClause): in this way, the configuration scope is the entire query and so the input elements can be used in every graph pattern without any restrictions [35]. On the other hand, CQELS specify a window configuration for every specific stream graph pattern (StreamGraphPattern) of the query, for this reason, the window definition is specific and well defined for every graph pattern [44]. This configuration is necessary to work in a native approach, to specifically configure the DSMS streaming application operators that manage data streams on the fly, without transforming the information from dynamic to static. For this reason it is important

know which graph pattern of the query is referred to which input stream and which window configuration, in order to provide the correct data, and the correct window, to the correct operator of a streaming application. C-SPARQL does not need this specification, because it correctly selects the elements from the different input streams with the specified window configurations, and it temporarily saves all information, from different streams, together and with the static data. From this point on, it processes all data at the same time, as if they were all static [44,35].

However, in this way, the C-SAPRQL execution performances are reduced by the necessity to store all data as static information, since they need to be queried by a standard SPARQL engine. CQELS does not need to use this intermediate condition, and exploiting a specific window configuration for every graph pattern, it can manage data streams in a most performant way.

The management of timing is provided by C-SPARQL only with the function *timestamp* that can be used to bind the timestamps linked to a RDF stream with the specific variable into the graph pattern [35]. This function can filter conditions of the query in order to express constraints on Streaming Linked Data and to provide the consequentiality order of events in data streams. CQELS does not provide this kind of function, important to manage the timing of the data. It can be useful in a smart city domain, if we want to detect all cars that are passed at the point *A* and then have turned at the crossroad at the point *B*. In this scenario we need to use a *timestamp* function in order to distinguish the cars that from *A* go to *B*, from the cars that from *B* go to *A*. So we can use as the constraint to distinguish in the first case, where the timestamp bound to cars at the point *A* will be smaller then the timestamp bound to the same cars at the point *B* [35].

The query registration is a feature of C-SPARQL useful in order to register the query into the DSMS with a specific name [35], to manage multiple continuous queries that are executing into the DSMS.

Multiple input streams can be specified in both C-SPARQL and CQELS, however managed in different ways: C-SPARQL specifies input streams into the input stream clause (FromStrClause); CQELS define a specific input for every specific graph pattern (StreamGraphPattern). These differences are driven by the different approaches exploited by C-SPARQL and CQELS. In the same way as in window configuration, C-SPARQL input are generic and valid into all query, while CQELS inputs stream are specifically to the graph pattern where they have been defined

[44,35]. The limitations of CQELS can be overcome by eliminate the strictly bound between input stream and graph pattern, even if the bound need to be conserved between graph patterns and window configurations.

CSV input streams are not provided by any SPARQL extensions, however they can be used to specify a CSV streams as inputs of the query. In order to express a CSV stream as input, it is necessary: a URI reference of the stream, a window configuration, and an optional parameter that specifies the number of field, where is contained the timestamp of the tuple. This parameter is necessary because every CSV input can present an undefined number of fields, so we need to specify where is placed the timestamp.

CSV graph patterns are features that, in the same way as the previous one, are not provide by any other SPARQL extensions. They allow binding fields of a CSV data stream with variable definitions. Since lines of a CSV does not present a RDF triples (subject, predicate, and object) every variable has to be manually binds to every field of requested CSV records, by exploiting a special RDF triple, used to define a single variable. This special RDF triple following the standard of a triples pattern, however it present a fixed configuration.

LANGUAGE FEATURES	QCELS		C-SPARQL	
Window concept	✔	Specified into *StreamGraphPattern*	✔	Specyfied into *FromStrClause*
• Time-based window	✔	•	✔	Specified with RANGE and STEP
• Triple-based window	✔	Specified with TRIPLES	✔	Specified with TRIPLES
• Window for graph pattern	✔	Specified into StreamGraphPattern with limited graph pattern scope	✖	Specyfied into *FromStrClause* with general scope
Management of timing	✖	No function to manage timestamps	✔	Managed with the function *Timestamp*
Query registration	✖	No operator to manage the query registration	✔	Specified with REGISTER STREAM or QUERY and COMPUTED EVERY
Multiple input streams	✔	Specified in multiple *StreamGraphPatterns*	✔	Specyfied in multiple *FromStrClauses*
CSV input streams	✖	No function to manage heterogeneous inputs	✖	No function to manage heterogeneous inputs
CSV graph patterns	✖	No function to manage heterogeneous graph patterns	✖	No function to manage heterogeneous graph patterns

Figure 3.6 Comparison between C-SPARQL and CQELS extensions **[44,35]**

It specifies the variable name as subject of the triples, a special predicate to identify the correct filed of the CSV line, and then a URI reference of the specific CSV input to which is referred the variable. By exploiting this construct, the variable can be later used as normal variables in the query execution.
Figure 3.6 summarises the capacities of C-SPARQL and CQELS, in order to have a graphical representation of the different capabilities of both languages.

3.3.2 New SPARQL extension definition

This section presents the formal definition of our new SPARQL extension that we name *DubExtensions* since is based on both C-SPARQL and CQELS specification. In the figure below are expressed all features that we have added in order to extend the SPARQL grammar: part of the grammar corresponds to C-SPARQL and part to CQELS, however, we further extend their features to support CSV graph patterns and CSV input streams (Figure 3.7).

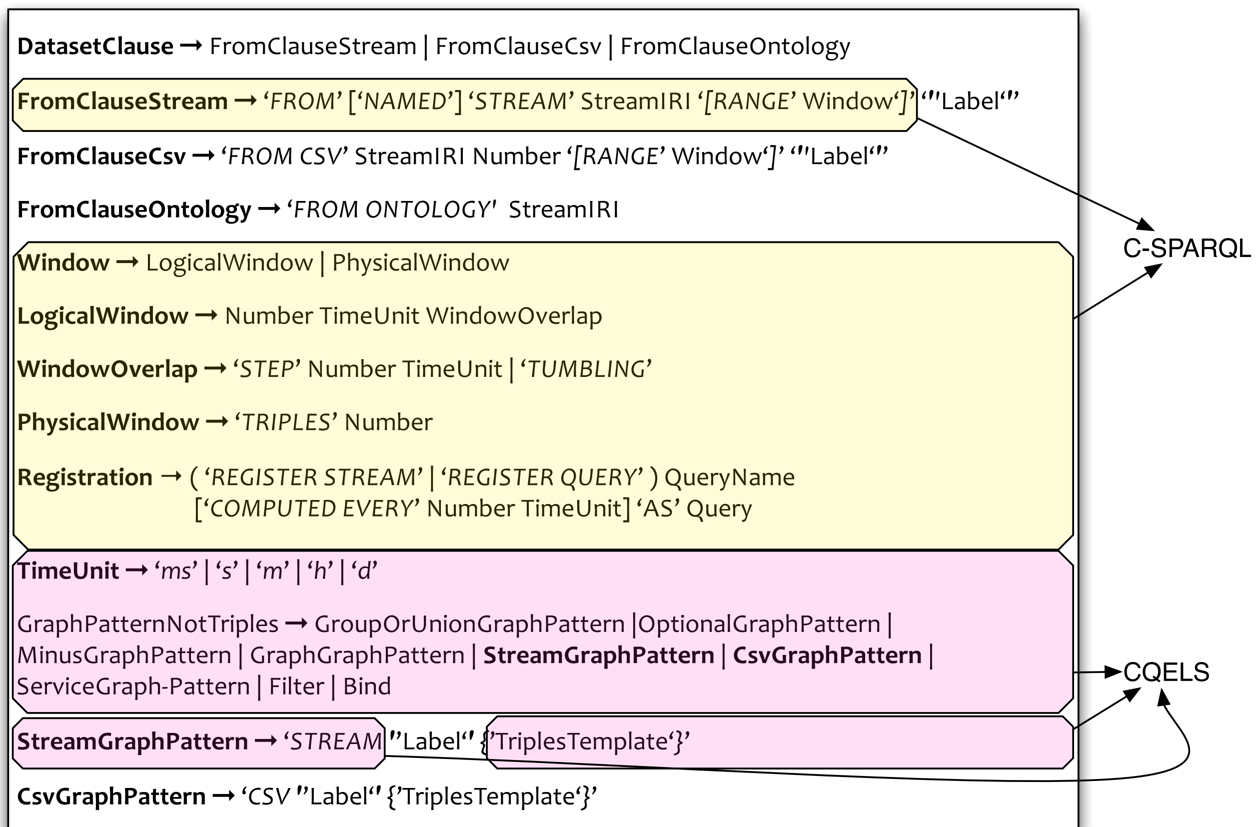


Figure 3.7 *DubExtensions* SPARQL extension definitions

In the language specifications, we have introduced the concept of label that can be useful in order to bind a specific input and the relative window configuration with the specific graph patterns. In this way, we are able to define an input stream specification with a scope that includes the entire query, so that it can be specified once for all possible graph patterns. So in this way, we overcome the C-SPARQL limitation, since our input stream configuration can be bound to a specific graph pattern. Moreover, note that, the concept of label is not bound to any specific input stream, but the same label can be shared between different inputs, even if they need to provide the same window configuration. That is necessary because, even if we can merge different input streams, we cannot provide different window configurations to the same graph pattern.

As one can see in Figure 3.7, we exploit the *StreamGraphPattern* extension defined by CQELS, even if we replace the input stream specification *(VarOrIRIref)* with the concept of label. Moreover in the same way we define the *CsvGraphPattern*, identified by the *CSV* keyword. As we have already said, the concept of label can be shared between different input streams, with the same window configuration, so it is possible to specify multiple input streams to the same stream or CSV graph patterns.

The stream graph pattern (StreamGraphPattern) does not present any change from the CQELS definition, with the exception for the concept of label; on the other hand, the CSV graph pattern (CsvGraphPattern) is completely new. The CsvGraphPattern does not have to deal with RDF stream and for this reason its triples pattern is fixed and designed in a specific way. Into the CsvGraphPattern, every triple pattern is used to define a single variable of the query. As we have said, the first element of the triple is the variable name that will contain 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 correct field of the CSV record, and the last element is the URI reference of the specific CSV input stream. To better understand the characteristics of stream and CSV graph pattern, one can see the two examples below (Figure 3.8).

StreamGraphPattern

```
1  STREAM 'stream1'{
2       ?sob :pred ?obj .
3  }
```

CsvGraphPattern

```
1  CSV 'csvstream' {
2       ?var1 ibm:csvCol_0 <http://.../example1> .
3       ?var2 ibm:csvCol_2 <http://.../example2> .
4  }
```

Figure 3.8 *DubExtensions* stream and CSV graph pattern examples

The example of StreamGraphPattern in Figure 3.8 uses the label `'stream1'` (Line 1 of Figure 3.8), to specify the input stream and the window configuration of the specific graph pattern. The example of CsvGraphPattern uses the label `'csvstream'` (Line 1 of Figure 3.8), and in this case, the graph pattern is expressed in a well-defined form, following the guidelines that we have previous defined. For example the variable "`?var1`" is defined as field zero ("`ibm:csvCol_0`") of the CSV stream identified by the URI *http://.../example1*.

Furthermore we extend the possible input specifications (*DatasetClause*) of SPARQL with three new elements (Figure 3.7):

- *FromClauseStream*: it specifies RDF stream inputs, and it is realised similar to the stream input specification of C-SPARQL, we the addition of the concept of label in order to bind a specific input to specific graph patterns. This clause is specified by the *FROM STREAM* keyword;
- *FromClauseCsv*: it specifies CSV stream inputs and it is similar to the FromClauseStream clause with new parameter a number, to specify the field of the CSV stream that contains the timestamp. This clause is specify by the *FROM CSV* keyword;
- *FromClauseOntology*: it specifies a static ontology as input, in order to work on stream reasoning. It represents a useful feature in order to specify that a specific static input, need to be shared between static and dynamic reasoner This clause is specify by the FROM ONTOLOGY keyword.

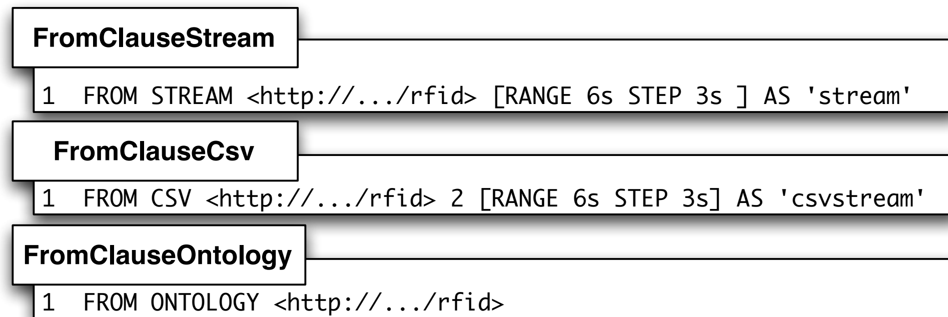


Figure 3.9 *DubExtensions* input specifications

All defined input specifications most work on both RDF and CSV data streams, and are responsible of the window configuration of every specific stream. We exploit the window definitions of C-SPARQL, with *RANGE* and *STEP* keywords to specify the logical window and *TRIPLES* keyword to specify the physical one, as one can see in the example above (Figure 3.9), where are shown all three possible input specifications.

Finally, to register the query to specific DSMS we also reuse the registration clause of C-SPARQL (Figure 3.7).

```
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://.../rfid>.
13      ?bikecount ibm:csvCol_2 <http://.../rfid>.}
14   } GROUP BY ?station ?stationid ?stationlat ?stationlong ?address
```

Figure 3.10 Example of *DubExtensions* query

The above example (Figure 3.10) represents a possible usage of a query *DubExtensions,* able to merge static RDF and stream CSV data stream. As one can see, the *DubExtensions* query provides the average of usage of bike for every possible station every 20 minutes, by selecting the static information about stations

("`?station`") from a standard Basic Graph Pattern, and by counting the number of bikes available ("`?bikecount`") from the CSV data stream.

This chapter has provided the complete and formal definition of our new SPARQL extension: *DubExtension*, from now on, we will exploit this language in order to define all necessary queries in our system.

4 DUBEXTENSIONS EXECUTION FRAMEWORK

Starting from the new SPARQL extension of the previous chapter, we aim exploiting InfoSphere Streams of IBM as execution framework for data stream processing. In order to do that, we need to provide a translator that from DubExtentions queries is able of producing streaming applications, by exploiting the SPL program language of InfoSphere Streams. For this reason the translator has:

- to parse and to interpret DubExtentions queries by their query string representations, providing an internal representation (a query object) that can be better manage by the system;
- to transform the query object that represents the DubExtentions continuous queries in the effective SPL streaming application, so that we can exploit InfoSphere Streams as query engine to execute the streaming applications.

We implement the translator based on the ARQ architecture, by exploiting the parser and analyser components, and by extending the ARQ with new component to provide the complete translation. In this way, our DubExtentions execution framework uses a "*white box*" approach as CQELS, because we generate streaming applications exploiting native operators of InfoSphere Streams and analysing data stream on the fly, without any intermediate translation. Moreover, we investigate on the integration of heterogeneous streams, and in powerful system for backward reasoning on data streams (Chapter 5).
First of all, in this chapter, we introduce the InfoSphere Streams operators (Section 4.1), and then we deal with the translation (Section 4.2), showing also some example of complete translation (Section 4.2.3).

4.1 STREAMS PROCESSING LANGUAGE OPERATORS

We have already presented SPL language but now we focus on its standard toolkit operators and their functionalities that can be relevant to execute DubExtentions continuous queries.

SPL presents several operators, also called node of the streaming application, where every node is able to manage multiple input and output streams, and with an associated *schema* definition that contains the set of data types that compose the stream. The stream are consists of discrete items, tuples, in SPL are defined by the same name types "*tuple*", that is in case, are defined as a sequence of attributes also known as named value pairs that are used to express the data [41]. The tuple `{name="Tom", age=32}`, is an example of this representation that presents two attributes, specifically `name="Tom"` and `age=32`.

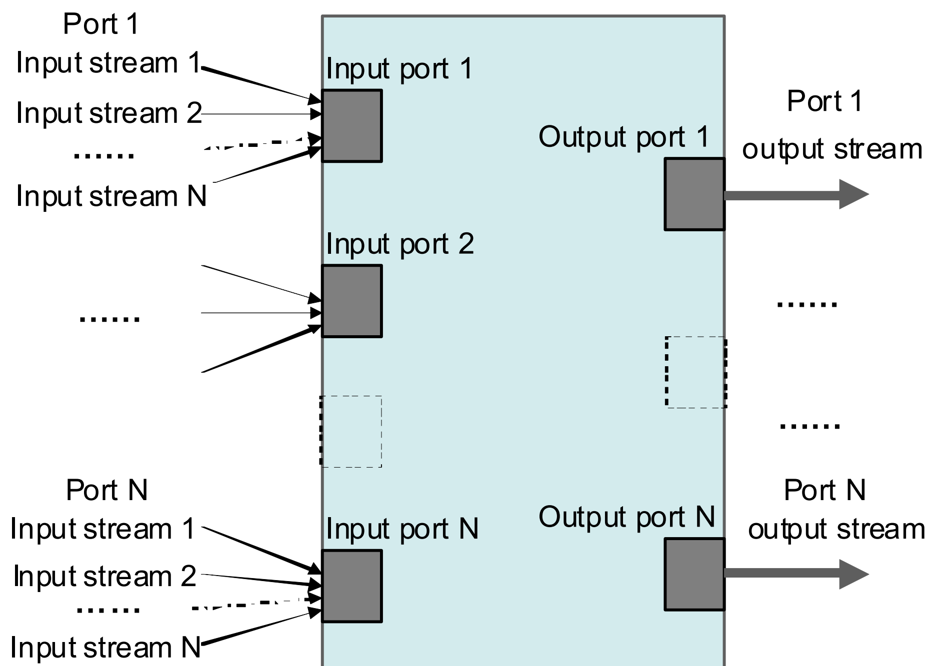


Figure 4.1 Generic SPL stream node **[41]**

Figure 4.1 shows all possible connections applicable to an SPL operator. For every node it is possible to specify several input/output streams that are connected to several input/output ports, which can be shared by multiple streams that present the same schema configurations.

SPL provides multiple primitive operators, used to implement streaming applications, that can be divided in three different classes, based on their specific usage: *Relational*, *Adapter*, and *Utility* operators [41].

4.1.1 Relational Operators

These SPL operators consist of a set of operators to realise fundamental functionalities. They are the same that can be found on standard relational databases, but in this case, they deal with stream information, so they need to consume the data on the fly, by exploiting the concept of window. Even if not all of them need to provide a window, but stateful operators do because their outputs depend on the previous data into the stream. In the two later subsections we are going to formally present both:

- *stateless operators*: they do not exploit the concept of window;
- *stateful operators*: they need to exploit the concept of window.

4.1.1.1 SPL stateless operators

The relational SPL stateless operators are [41]:

- *Filter;*
- *Functor;*
- *Punctor*.

The **Filter** operator performs the selection of tuples on the stream. It can be configured with user-specified conditions that the specific tuples have to satisfy in order to pass through the filter (Figure 4.2). Therefore it is a stateless operator, because its execution does not depend by previous tuples of the stream, and for this reason, the Filter does not accept any window configuration [41]. In the example below on can see, a Filter operator filtering a stream of input tuples greater than 12 and producing a subset of the stream with the results.

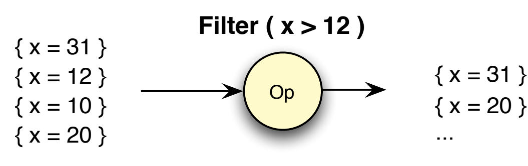


Figure 4.2 Example of SPL Filter operator

The **Functor** and **Punctor** operators perform the projection and rename of tuples from the input to the output streams. The two operators consider as input, tuples with a specific set of attributes, and they produce as output, specified transformations of the input tuples by reducing, increasing, and renaming the input

tuples [41]. The Punctor unlike the Functor operator presents an additional functionality; it can add markers (punctuations) into the output stream. In this way, the Punctor operator can be used to manage the window definition into the stream, and to specify that a window is complete and a new window can start.

Both Functor and Punctor operators are stateless, and for this reason, they do not have any window configuration. In the example below one can see a Functor and a Puntor operator projecting and renaming into the output stream, the only user-specified attributes.

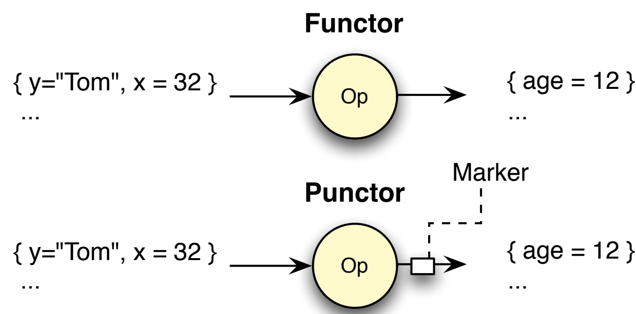


Figure 4.3 Example of SPL Functor and Punctor operators

4.1.1.2 SPL stateful operators

The relational SPL stateful operators are [41]:

- *Sort;*
- *Join;*
- *Aggregate.*

They are stateful because, they need to select a specific set of items from the stream, in order to calculate their outputs, since they depend by all tuples selected by the window, from the stream.

The **Sort** operator aims of ordering tuples, on the base of user-specified ordering expressions. The Sort is a stateful operator, and for this reason, it needs to identify a finite portion of the unlimited data stream by exploiting the concept of window, and then order all elements inside the window. The sorting process is executed when the window is defined “full” and for this reason until that moment the Sort does not put in output any tuple. Only, when the window is “full”, all tuples are pushed out simultaneously. In the example below the operator alphabetically orders a set of tuples of the stream that appear into the window configuration.

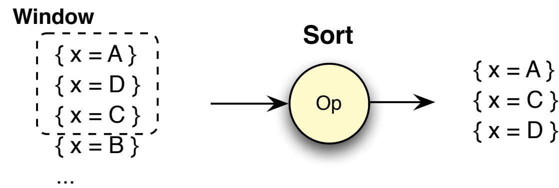


Figure 4.4 Example of SPL Sort operator

The **Join** operator is used to correlate tuples from two streams based on user-specified match values; it is a stateful operator that adopts two window configurations for both input streams, *left* and *right* inputs [41].

When a tuple is received on an input port, it is inserted into the corresponding window, which in this way is triggered and start to process the coming tuple. As part of the trigger processing, the tuple is compared against all tuples inside the window of the opposing input stream. The comparison is made on user-specified values that have to be present in both windows. Therefore for every match that is found, an output tuple will be produced [41]. This particular approach represents the *Inner join (join)*, where the tuple, coming from the left or the right side of the join, has to match into the opposite one.

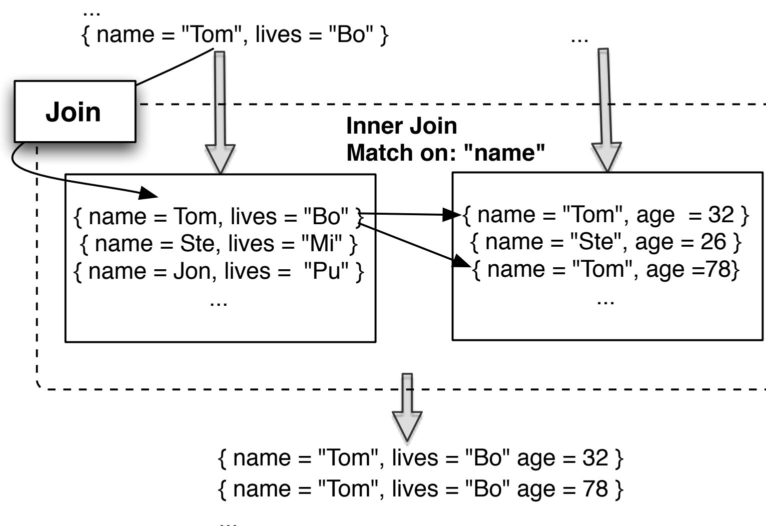


Figure 4.5 Example of SPL Join operator (*inner join*)

Figure 4.5 presents an example of *Inner Join*, where the tuple `{name="Tom",lives="Bo"}`, coming on the left port of the join, it is stored into the current left window, and at the same time the window triggers the process of inner join. In this way the tuple is compared with the right window of the join, finding two

matches on the attribute "name". Thus the matching tuples are combined together and pushed out, without need to wait that the window is "full" [41]. However if there is no user-specify attribute to match, the join will look for the match in all attributes, and it will perform a *Cartesian product* of all matching tuples, as one can see in the example in Figure 4.6.

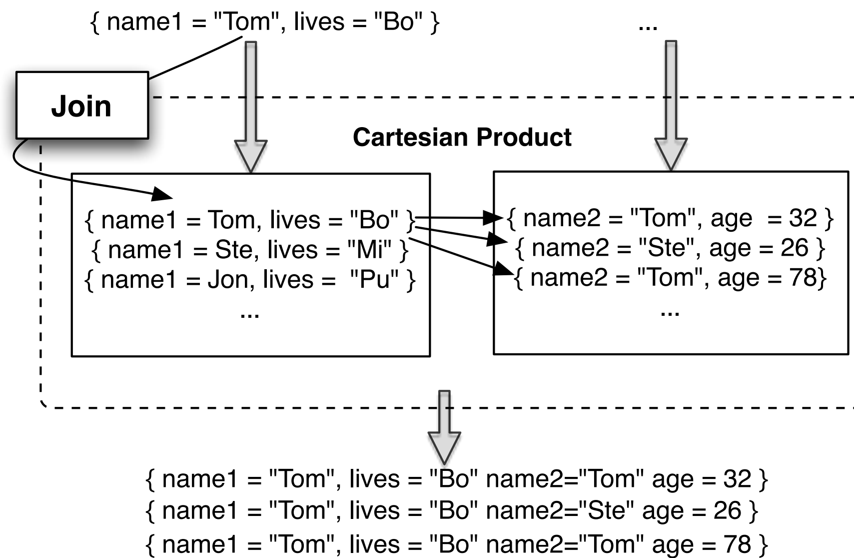


Figure 4.6 Example of SPL Join operator (*Cartesian Product*)

There are other possible join algorithms: *left outer/right outer* and *outer* (Figure 4.7). The left outer/right outer (or simply left/right) work in the same way, even if they are one the opposite of the other. The left join gives "priority" to the left port of the join and in this way, all the tuples that are coming into the left port will be pushed out when the window is "full" even if they do not have any match on the right side. If there is no match the missing attributes of the tuples are substituted with default values. Note that, in this case the tuples hold back until the window is "full" to come out, since a matching tuples could come at any time [41]. The right join work exactly in the same way, even if it gives "priority" to the right port of the join.

Usually the join operator provides two input ports, for coming data, and one output port, for the results. However working on left/right algorithms is possible to specify an additional port. The second output port contains the right/left incoming tuples that have no matches into the opposite side. For example a left join has a tuples on the right port that does not have match into the left mandatory side; when the window is "full" it will out on the second output port. This functionality can be very useful in

order to know which tuples do not match the existing ones and it is also called anti join [41].

The last algorithm is the outer join, that is a combination of the left and right joins, so in this case all the tuples will come out with no matter to match the left or the right side of the join [41]. An example of left, right and outer joins are shown below (Figure 4.7).

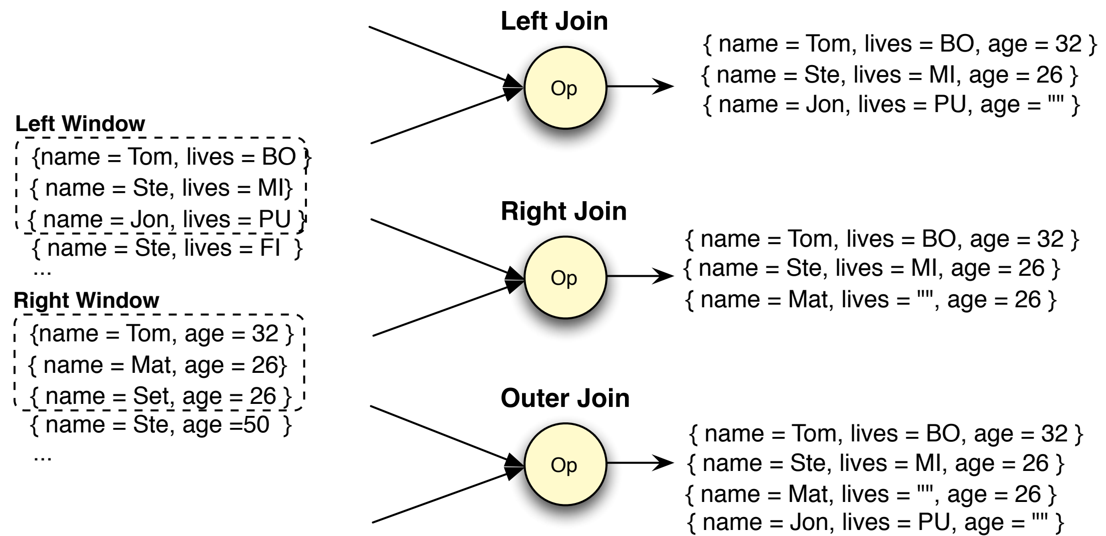


Figure 4.7 Example of Left/Right/Outer Join

The **Aggregate** operator is used to compute user-specified aggregations over tuples, for this reason it is stateful and it needs to adopt a window configuration to select tuple from the stream. The aggregate operator can be used to specify one or more expressions to be used for dividing the selected tuples of the window in multiple groups [41], where is also possible to apply additional functionalities like sum/min/average, and etc. [41]. The aggregate operator usually waits until the window is “full” to push out all results, however it is possible to push out the values beforehand, by forcing the window configuration with a sliding parameter.

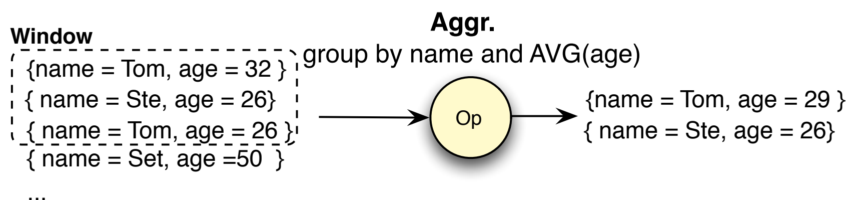


Figure 4.8 Example of SPL Aggregate operator

In the above example one can see a possible Aggregate operator that groups the tuples by name, and it calculate the average of the ages of every groups.

4.1.2 Adapter Operators

Adapter operators are a set of operators that is composed by sources and sinks of streaming applications. They are used to manage the heterogeneous input and output stream formats that can be obtains from different sources. Moreover, they are stateless operators, because they cannot be configured with any window specifications. We are interested in adapter operators that can be used to read and write files, since the stream information, which we have, are physically stored on files [41]:

- *FileSource;*
- *FileSink.*

The **FileSource** operator is used in order to read data from a file and produce tuples as results. It is usually configure with a single output port in order to read different formats and produce output tuples with specific schema configuration.
The **FileSink** operator is used in order to write tuples into a file. Contrary to the FileSource operator it usually configured with a single input port.

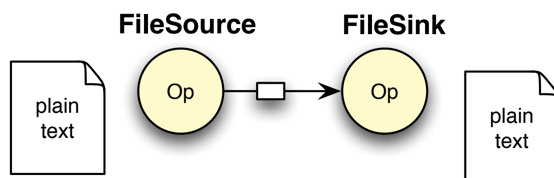


Figure 4.9 Example of SPL FileSource and FileSink operators

In the example above one can see, the FileSource operator reading tuples from a file, and the FileSink that is writing the same tuples into another file.

4.1.3 Utility Operators

Operators that can provide a large set of functionalities compose Utility Operators. Here we are interested in presenting two fundamental operators we have exploited in the translation from DubExtentions to SPL [41]:

- *Custom;*
- *Beacon.*

Both these operators can be considered stateless because, even in this case, it is no possible to specify any window configuration.

The **Custom** operator is a special operator provides by SPL that can receive and send any number of streams and does not do anything by itself. It represents an operator that can be completely configure, "*It offers a blank slate for customization*" [40] and in this way, very useful to provide a high level of customization and realise specific functionalities. In the example below, the Custom operator is dividing three different inputs in other two streams, based on the "`bus`" number.

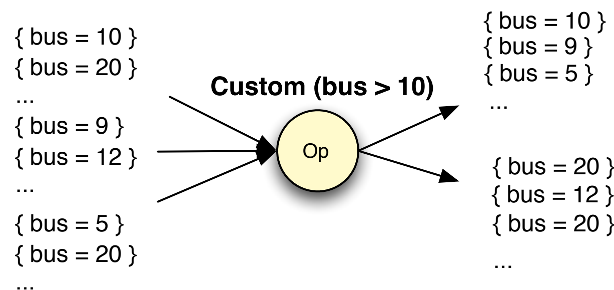


Figure 4.10 Example of Custom operator

The **Beacon** operator can be used as utility source, in order to generate tuples on the fly, with a fixed schema. It is a stateless operator and moreover it does not have any input ports, but just one output port. The example below shows a Beacon node used to generated a stream with "`{bus = INTEGER}`" as schema.

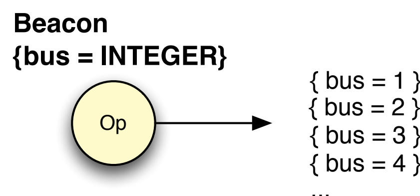


Figure 4.11 Example of Beacon operator

4.2 LOGICAL TRANSLATION OF DUBEXTENSIONS QUERY TO SPL

This section presents the effective translation from DubExtentions query into SPL streaming application: starting from the definition of DubExtentions that we have provided of this extension (Chapter 3), we are able to express continuous queries in a query string form. In this way, queries define a high-level specification for a SPL

streaming application that will execute on InfoSphere Streams. However this translation is not so trivial, because the two languages work at two complete different levels of specifications. DubExtentions is a powerful language used to express high-level concepts without concern on how this data are retrieved from the stream. While SPL is concerned about how build streaming applications and how connect different operators (nodes) together, and even if SPL is a declarative language it is similar to a physical language, used to produce the specific data flow to process the information, with nodes that can perform simplest binary and unary relational operations.

In order to fill the gap between the two languages, we decided to exploit a two phases transformation:

- *the ARQ translation*: it is the first translation where, starting from the DubExtentions query string we exploit the ARQ parser and the ARQ analyser models [47], in order to obtain an intermediate representation of the query string as query object;
- *the SPL translation*: it is the second translation where, we work on the intermediate representation to transform the query object into the SPL streaming application.

Because ARQ does not accept any SPARQL extension, we extended its parser and analyser, adding all features presented in the previous chapter (Chapter 3), in order to support DubExtentions query strings. In this way, ARQ becomes able to produce the exact translation of the DubExtensions query string, in a query object. We will not present the extension of ARQ parser and analyser in this dissertation, however we will analyse the results of the translation, in order to understand the simplification provides to a query string during this phase (Subsection 4.2.1). Then we will focus our attention on the SPL translation: from the query object produce by ARQ to the SPL streaming application, during the second phase (Subsection 4.2.2).

4.2.1 First step: the ARQ translation

During this first phase of translation, we use an ARQ architecture to provide an intermediate representation of DubExtensions query strings by exploiting the SSE language, based on SPARQL Algebra and used to provide the exactly execution steps of a SPARQL query. The parser and the analyser ARQ module are able to

produce and internal representation of a query by exploiting the SSE language [47]. However, even if they can manage SPARQL queries, they cannot manage our DubExtensions language; we have extended the ARQ capabilities in order to support all new features provided by our extension. However because DubExtensions is an extension of SPARQL, the core of its translation is represented by the translation of SPARQL.

4.2.1.1 Standard SSE translation of a SPARQL query

This represents one of the most important phase during the ARQ query execution, because the order sequence of steps that have to be used in order to evaluate the query are fixed, by expressing the SPARQL query with chains of simpler operators [47]. We want to exploit this ARQ capability, because it can provide a simpler query form, useful to reduce the gap between DubExtensions and SPL.
With this internal representation of a SPARQL query, ARQ exploits SPARQL S-Expression, its internal language. In this way, ARQ represents a SPARQL query as a chain of nested operators that have to be computed in a *bottom-up approach*, starting from the more internal one and proceed in a backward until the more external [47].
The most effective simplification of a query SPARQL is to provide the query pattern, while all components of the query, with some exceptions, are translated with a one-to-one correspondence. For this reason there is no introduction of any simplifications. However for sake of completeness, we present all operators that have been necessary in our use cases, in order to translate DubExtensions query, and those that one will find into the next phase of the translation to SPL (Subsection 4.2.2).

The Triple pattern and Base Graph Pattern translation

The simplest operator defined by SSE is “`triple`” that can contain just a single triple pattern, where all the Turtle shortcuts have been unfolded, and that does not contain any other operators, representing the bottom level of the chain, as one can see in the example below [47].

```
(triple ?sub <http://completeURI#pred> ?obj)
```

The triple pattern is always contained into another basic operator: the “`bgp`” operator, which as suggested by its name, it represents the Base Graph Pattern of a

SPARQL query, where is possible to specify multiple triples that need to match simultaneously, in order to select specific braches of RDF graph. In the example below one can see how is realised the construct of a simple "bgp" operator that contains two triples patterns [47].

```
( bgp
    (triple ?sub <http://completeURI#pred> ?obj)
    (triple ?sub <http://completeURI#pred2> ?obj1) )
```

The "bgp" operator is a basic operator because it is the only operator that can manage triple pattern, and for this reason it represents the last level of graph pattern into the chain of nested operators. In the example below, one can see a possible translation of a Basic Graph Pattern, composed by two triples, in SSE operators.

Basic Graph Pattern	*SSE Basic Graph Pattern*

WHERE{
?sob :pred ?obj
?sob1 :pred1 ?obj
}
	(bgp
	(triple ?sub :pred ?obj)
	(triple ?sub1 :pred1 ?obj)
	)

Table 4.1 Example of SSE Basic Graph Pattern translation

Graph Patterns operators

All the graph patterns, with the exception of the BGP, are based on *binary operators*, that can operate just between two elements at a time, that represents an important simplification, that provide a fixed evaluation steps, as one can see in the example below, where a binary operator is joining three different BGP in a specific order.

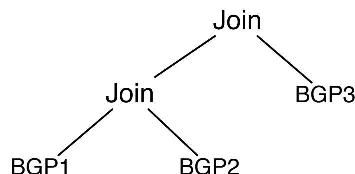


Figure 4.12 Example of binary operator performing a Join on three BGP

We analyse the graph patterns of SPARQL and we provide an exemplificative translation by exploiting SSE operators, specifically we will present the translation of [47]:

- the *Group Graph Pattern;*
- the *Optional Graph Pattern;*
- the *Alternative Graph Pattern;*

The **Group Graph Pattern** expresses the join of multiple graph patterns and the Join operator of SSE, identified by the "join" keyword, is used to translate it. However because it is a binary operator, it needs to realise a chain of joins in order to translate more the two input elements (Figure 4.12). Considering the example below where there are three "bgp" operators, that are composing the Group Graph Pattern, it is necessary to join two of them, and then to join the result with the third one, following a distributive approach (Table 4.2).

Group Graph Pattern	*SSE Group Graph Pattern*

`WHERE{`
` {?sob :pred ?obj}`
` {?sob1 :pred1 ?obj}`
` {?sob2 :pred2 ?obj}`
`}`
	`( join`
	`  ( join`
	`    ( bgp (triple ?sub :pred ?obj) )`
	`    ( bgp (triple ?sub1 :pred1 ?obj) )`
	`  )`
	`  ( bgp (triple ?sub2 :pred2 ?obj)`
	`)`

Table 4.2 Example of SSE Group Graph Patter translation

The **Optional Graph Pattern** as the Group Graph Pattern, is translated by joining multiple graph pattern together, even if in this case, to do that it is used the left join operator of SSE, identified by the "leftjoin" keyword. The left join is used to express that the left size of the join is mandatory and for this reason it can be part of the query results even if there is no match with the right part, which is optional. In the table below one can see an example of SPARQL Optional Graph Pattern and its translation with SSE operator (Table 4.3).

Optional Graph Pattern	*SSE Optional Graph Pattern*

`WHERE{`
` {?sob :pred ?obj}`
` OPTIONAL{?sob1 :pred1 ?obj}`
`}`
	`( leftjoin`
	`    ( bgp (triple ?sub :pred ?obj) )`
	`    ( bgp (triple ?sub1 :pred1 ?obj) )`
	`  )`

Table 4.3 Example of SSE Optional Graph Patter translation

The **Alternative Graph Pattern** is used to unify multiple graph patterns together, and the Union operator of SSE that is identified by the "union" keyword is used to translate it. In this case as for the join operator, the "union" produces chains of union operators to translate more then two elements together. The table below shows

an example of SPARQL Alternative Graph Pattern and the relative SSE representation.

Alternative Graph Pattern	*SSE Alternative Graph Pattern*

`WHERE{`
` {?sob :pred ?obj}`
` UNION{?sob1 :pred1 ?obj}`
` UNION{?sob2 :pred2 ?obj}`
`}`
	`( union`
	`   ( union`
	`      ( bgp (triple ?sub :pred ?obj) )`
	`      ( bgp (triple ?sub1 :pred1 ?obj) )`
	`   )`
	`   ( bgp (triple ?sub2 :pred2 ?obj)`
	`)`

Table 4.4 Example of SSE Optional Graph Patter translation

All these graph patterns can be further combined to realise more complex patterns, and in this way completely translate a query pattern of a SPARQL query string.

Other operators

Going back along the chain of nested operators we have a set of operators that can be used to modify the query pattern results [47]:

- the *filter clause*;
- the *query modifiers*;
- the *query result clause*.

The **filter** clause is translated with the filter SSE operator that is identified by the "`filter`" keyword, used to expresses algebraic and logical conditions in order to filter the results.

Filter Clause	*SSE Filter Operator*

`WHERE{`
` ?sob :pred ?obj.`
` FILTER(?sob > 10)`
`}`
	`( filter (> ?sob 10)`
	`  ( bgp (triple ?sub :pred ?obj))`
	`)`

Table 4.5 Example of SSE filter operator

The **query modifiers** are usually inserted before the query result clause, with the exception of *slice*, that is translated with the "`slice`" SSE keyword and that is the last operator of the chain.

All other modifiers have to necessary perform before the projection clause *(SELECT),* because their may express conditions on variables that are not going to

be into the projected results set. The *ORDER BY* clause and the *GROUP BY* clause are translated respectively with the “`order`” and “`group`” SSE operators.

Slice/Order/Group by Clauses	*SSE slice/order/group Operators*

`...}`
`GROUP BY ?sob`
`ORDER BY ?sob`
`LIMIT 10`
	`(slice _ 10`
	`  (order(?sob))`
	`    (group (?sob))`
	`      ...)`

Table 4.6 Example of SSE slice/order/group operators

The **query result clause** performs the projection, the rename, and additional algebraic operations on the result set, and it is usually placed as last operator of the chain. This clause, depending on its complexity can be translated with “`project`” and “`extend`” SSE operators. The “`extend`” operator solves the algebraic expressions, and realise the renaming on the results. While the “`project`” performs the final projection of the variables into the results set, as one can see in the example below.

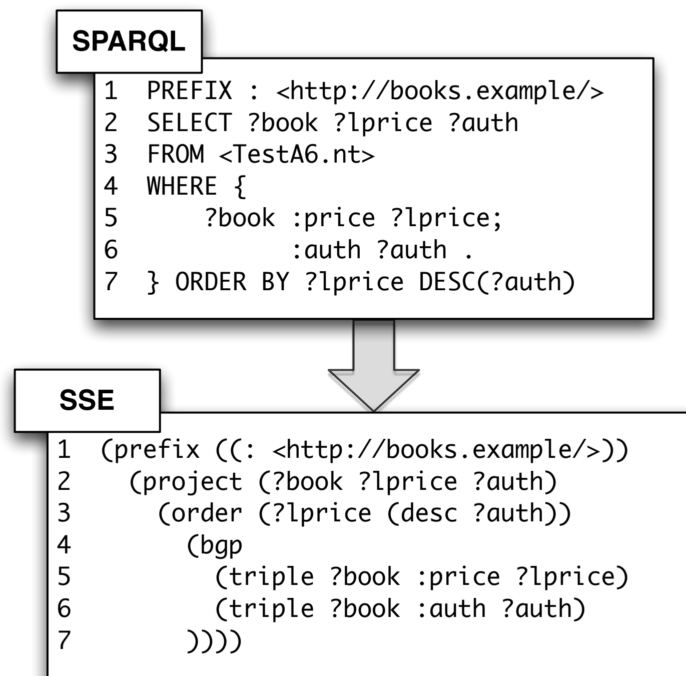


Figure 4.13 Example of complete translation of a SPARQL query to SS-Expression

Figure 4.13 shows a complete translation of a SPARQL query into SPARQL S-Expressions. However until now we do not introduce any new element to work on data streams, these elements will be presented in the next subsection below.

4.2.1.2 SSE translation of a DubExtensions query

In this section, we focus in translation of all elements of our DubExtensions extension. As a first note, the registration conditions and input specifications do not need to be translated, because they can be processed directly so they are not part of the nested chain of SSE operators. Considering again the DubExtensions definition (Figure 3.7), we have defined two additional Base Graph Patterns, which need to be translated with ARQ into query object:

- the *stream base graph pattern*: responsible to find matching variables on specific data stream;
- the *CSV base graph pattern*: responsible to bind its variable with specific fields of a CSV stream;

Moreover, in the same way as all graph patterns, they can be combined in different models, in order to compose the DubExtensions query pattern. Since both stream and CSV graph patterns can directly access to the triple patterns, they can be considered basic operators. In this way, as the Base Graph Pattern they can be translated with an extension of the "`bgp`" operator of SSE.

For this reason we define two extension of the "`bgp`" operator:

- *stream bgp;*
- *CSV bgp;*

The **stream bgp** is always associated with to specific input data streams by exploiting the concept of label that we need to preserve in the translation. Indeed the label value is inserted into the name keyword of the new stream bgp operator, which is identified by: "`stream_`" plus the label value plus "`:bgp`". Thus if, for example, the label value is "*wheather*" the complete stream bgp name will be: "`stream_wheater :bgp`".

The **CSV bgp** as the stream bgp, is always associated to specific input data streams by the concept of label. Therefore in this case, the keyword used to identified the new CSV bgp operator is composed by: "`csv_`" plus the label value plus "`:bgp`". Thus if, for example, the label value is "*bikestream*" the complete stream BGP name will be: "`csv_bikestream:bgp`".

In the example below, one can see a complete translation of a DubExtensions query that uses different types of graph patterns, and that gives an idea of how all the elements that we have presented work together (Figure 4.14). The translated query is

the same query that we have already presented in the previous chapter (Figure 3.10). However, here we focus on its translation into SSE operators. Moreover in this case, it is presented one of the two new defined bgp operators that we have defined, the CSV bgp: it works on CSV data streams, producing results that are joined together with the standard bgp operator. The SSE operator “group”, which is used to calculate the average usage of the public bikes for every station and manage the results of the join.

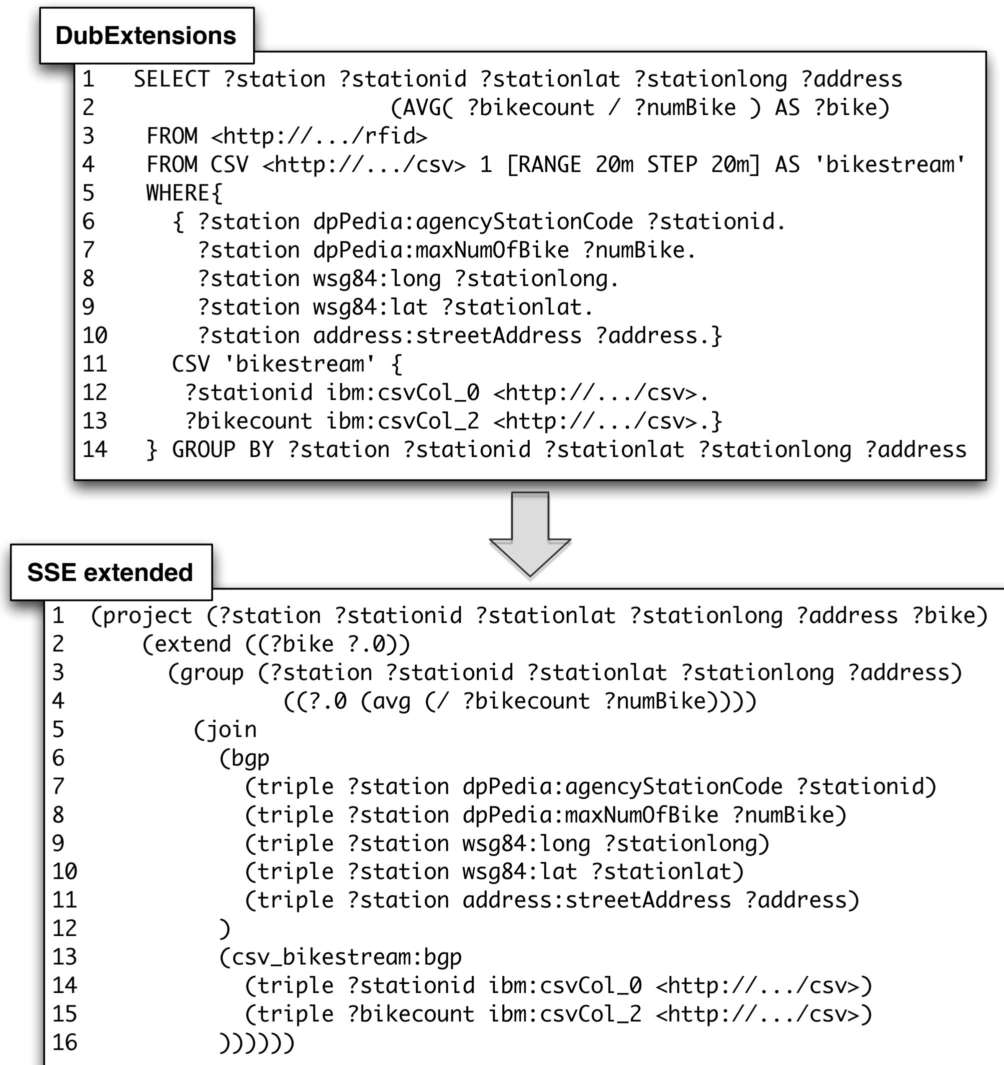


Figure 4.14 Complete example of DubExtensions first phase translation

At this point of the first phase, we have the complete translation of every possible DubExtensions queries in an extended version of SPARQL S-Expressions, and consequently we have reached the first simplification needed, in order to continue with the second phase: the SPL translation (Section 4.2.2).

4.2.2 Second phase: the SPL translation

The first phase reached the query object, an intermediate representation of the DubExtensions query, by exploiting the parser and analyser module of ARQ, which provide a simplified expression of the query with the extended version of SPARQL S-Expressions (with stream and CSV BGPs). This section presents the second phase of the translation by reaching the complete transformation of a DubExtensions query into SPL streaming application.

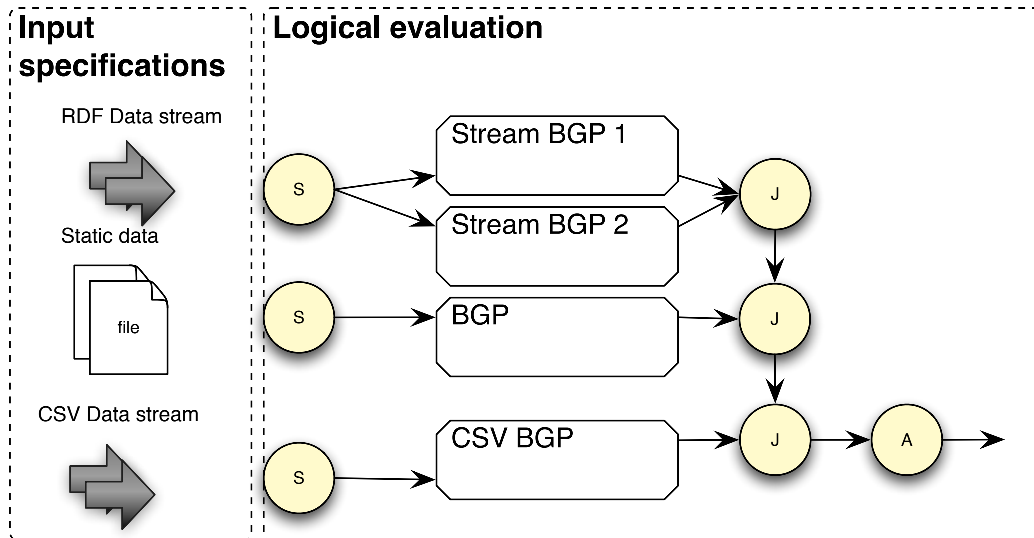


Figure 4.15 Overview of a SPL streaming application for a DubExtensions query

The above figure shows an overview of a possible streaming application, obtained from a DubExtensions query string, by managing multiple and heterogeneous input streams. The representation aims at giving a logical division of two possible modules to manage the translation of different part of the query:

- the *Input specifications module*: this part contains all possible heterogeneous input specifications defined by a DubExtensions query (Subsection 4.2.2.1);
- the *Logical evaluation module*: this part contains all nodes that are responsible for the evaluation. It corresponds to the query result clause, the query pattern, and the query modifiers (Subsection 4.2.2.2).

Another important part of a streaming application is the stream connections, used to provide links between different SPL operators. Those links need to be specifically configured by exploiting the triple pattern definitions of the query, where the variable names are used as attribute names of the tuples into the stream.

The last component necessary to produce a streaming application is the window configuration, of every stateful SPL node. We will deal with the window specification in the last part of this section (Subsection 4.2.2.3).

As we have already said, every SPL operator can also be considered as a node of the streaming application, so in this section we will refer to these operators as nodes, in order to not confuse the SPL nodes with the SSE operators (presented in the previous subsection).

4.2.2.1 Input specifications module

The *Input specification module* is the part of the query responsible of managing the input specifications of heterogeneous data streams by exploiting the clauses:

- *FROM* and *FROM ONTOLOGY;*
- *FROM CSV* and *FROM STREAM*;

The first two clauses refer to static inputs and are considered together in this section, as simple *FROM*, because they will be further analysed in the next chapter of this dissertation, where will address concept of stream reasoning.

The *Parser node* have been defined in order to manage all possible inputs: the node has been defined from the SPL custom node, in order to validate the inputs data and to tag every input tuples with the specific label value.

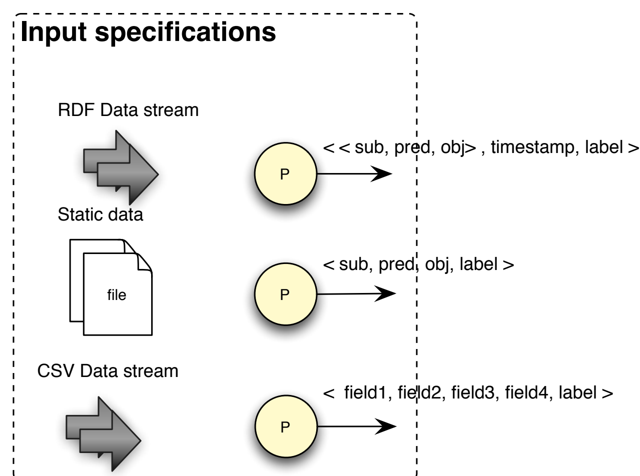


Figure 4.16 Logical representation of the Parser node and its tuples results

To every tuples is added an attribute called *label* that specifies its origin input stream and where it has to be routed. The three different input types, RDF, RDF stream, and CSV stream, are kept separated, in order to provide a specific stream reasoning functionalities on every single types of inputs.

In this way from this phase we have three different types of tuples, with different attributes as in Table 4.7 below.

RDF:	<< sobj, pred, obj >, label>
RDF stream:	<< sobj, pred, obj >, timestamp, label>
CSV stream:	< field1, field2,…,fieldN, label>

Table 4.7 Generic RDF, RDF stream and CSV stream specifications

Note that, even if the static input specification *FROM* has no label, we insert a fixed label value into the static RDF tuples, in order to have as much as possible homogeneous data specifications.

At this point of the streaming application, we do not have any information about the input tuples, thus we can just know that we have received well formed RDF, RDF stream triples and CSV records. Note that, even if for both RDF and RDF stream the input format is defined, it is not possible to previously know the number of attributes of a CSV record, because they can change based on the number of its fields.

4.2.2.2 Logical evaluation module

This module is the core of the streaming application that provides all necessary nodes to process the input tuples. As we have said, in this subsection we focus on the query translation without considering possible problems related to the window configuration.

To evaluate the query, we need to translate the query into a set of SPL nodes that have to correspond to the SSE operators previous defined, and in order to provide this kind of translation we specify three different types of classifications:

- *Auxiliary operators*: they represent SPL nodes that do not have any correspondent to any SSE operators, and they are defined to manage the routing of the tuples inside the streaming application;
- *Composite operators*: they represent all defined SSE operators that do not present a correspondence one-to-one with SPL nodes;
- *Single operators*: they represent all defined SSE operators that present a correspondence one-to-one with a single SPL node;

Note that, the Logical evaluation module takes place after the input specifications module, so we can assume that all coming tuples have been previous tagged with the

correspondent label, associated to the specific input stream, and that they are expressed as we have previously defined (Table 4.7).

Auxiliary Operators

Auxiliary Operators are all SPL nodes that do not present any relationships with the SSE operators, but they have been defined in order to manage tuples into the stream, starting from the SPL custom node and completely defining their functionalities. The auxiliary operator are composed by:

- the *Split node*;
- the *Union node*.

The **Split node** is responsible to split the input stream in multiples output streams by exploiting the label definition and the predicate specification of every single triple pattern, of the DubExtensions query, as one can see in the figure below.

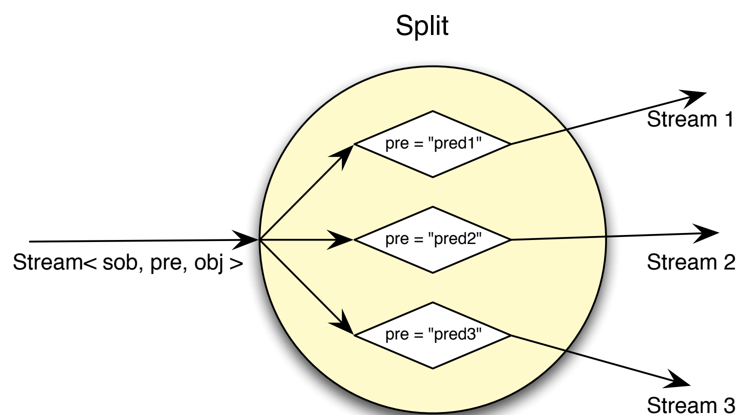


Figure 4.17 Auxiliary operator Split node

Thus, the Split node is represented as a sort of router that directs the tuples into their designed nodes of the streaming application, by exploiting two level of routing:

- *outside the BGPs*: in this case it is used the label attribute, in order to route the correct tuples into the correct Stream, CSV, or Standard BGP;
- *inside the BGPs*: in this case it is used the predicate attribute of the tuples to route them into the correct node inside the BGP.

In the example below one can see a logical representation of a Split node used to route two tagged RDF stream triples into two different stream BGP operators.

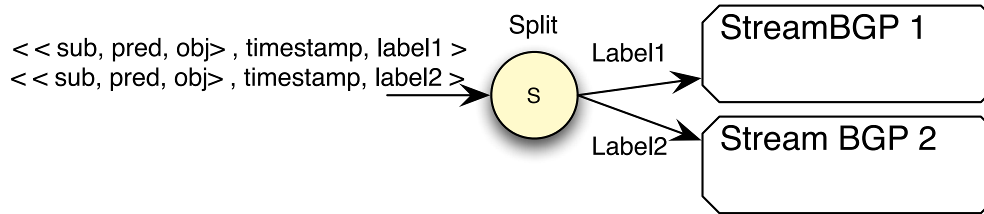


Figure 4.18 Example of outside Split node routing

The **union node** is responsible for the merge of multiple streams by fusing their attribute and providing a single output stream. However in this case, the tuples are not joined together, but they are used to fill all attributes of the output stream (that contains all possible attributes without replications), The missing elements of the output tuples are fill with default values, as in the example below.

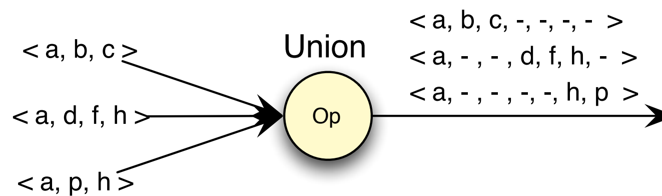


Figure 4.19 Auxiliary operator Union node

A particular configuration of the union node can also be used to translate the Alternative Graph Pattern expressed by the union SSE operator, as we will see later, however we have decided to highlight their difference usage by inserting the two nodes in two different classes. In this case, the union node is use to manage the streams and it can present an unlimited number of input streams, on the other hand, the union node that represents an alternative graph pattern is a binary node used to provide a logical union operation on the input streams.

Composite Operators

Composite operators are all SSE operators (with relative extensions) that are translated with multiple SPL nodes. In this case, primitive SPL operators are exploited at the same time with the Split node in order to translate this class of operators, that contains:

- *bgp SSE operator*;
- *stream bgp SSE operator*;

the CSV bgp SSE operator is not member of this group, because while the others are both working on standard triple patterns, the CSV bgp operator is completely different: it works on special triple patterns that needs to be translated in completely different forms, and whose translation can be performed with a single SPL node.

As we have said, both *bgp* and *stream bgp* operators are composed by triple patterns, and for this reason, they are expecting to find a match simultaneously for all contained triples, in order to select the information and send it to the next nested SSE operator in the chain. For this reason, both bgp can be translated in a single composed node of the graph that takes as input, all possible tuples that have been tagged with its corresponding label, and that provides as output all the subsection of input tuples that match all triple patterns.

In order to manage the concept of window, we define two possible BGP nodes: the *standard BGP* and the *stream BGP* nodes. However, because in this subsection we do not introduce the concept of window, we will refer to both standard BGP and stream BGP nodes with the unique acronym *BGP node*.

The **BGP node** is generated starting from the definition of bgp stream and bgp SSE operators as single input and single output composite node. Starting from the triple patterns that are defined in both bgp operators we are able to configure a chain of SPL Join operators that can provide the match for the all input tuples of the Base Graph Patters, as one can see in the example below.

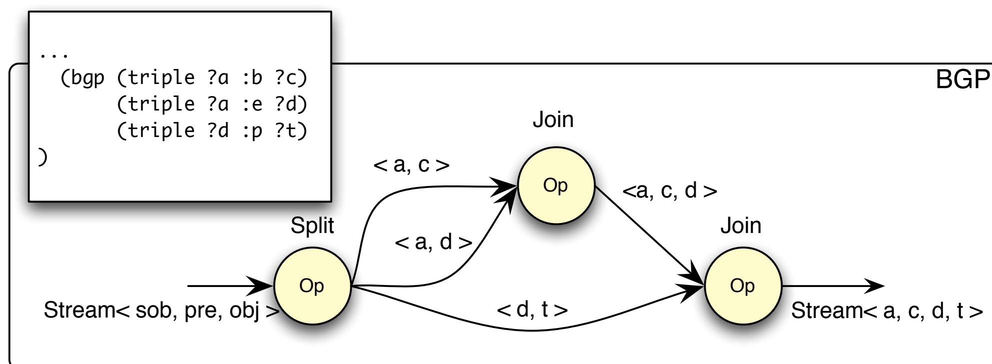


Figure 4.20 Example of composite Standard BGP operator

All the tuples that have been tagged with the corresponding label are flowing into the BGP node until the Split node, responsible to route the correct tuples into the correct Join nodes of the chain by exploiting the predicate element of the triples (Figure 4.20). In this way, it is defined a new stream connection for every triple

pattern contained into the BGP operator, and every coming tuples on the base of its concrete value attributes is routed into the right stream connection, which is just composed by attributes corresponding to its variable names. Moreover, all concrete values of the triple patterns are removed, because they are no longer necessary, since they cannot add any useful information in order to find the values of the variables; the concrete values are exploited by the Split node to complete its duty of routing and filtering the tuples from the input stream. Thus, the Split node feeds the chain of Join nodes to perform all necessary joins on the incoming tuples of the stream, by matching their specified attributes. In order to pass through the BGP node, a tuples need to match with all Join nodes of the chain that, for the exception for this first and the last ones, are sharing a stream connection: the output of a Join node became the input of another one (Figure 4.20).

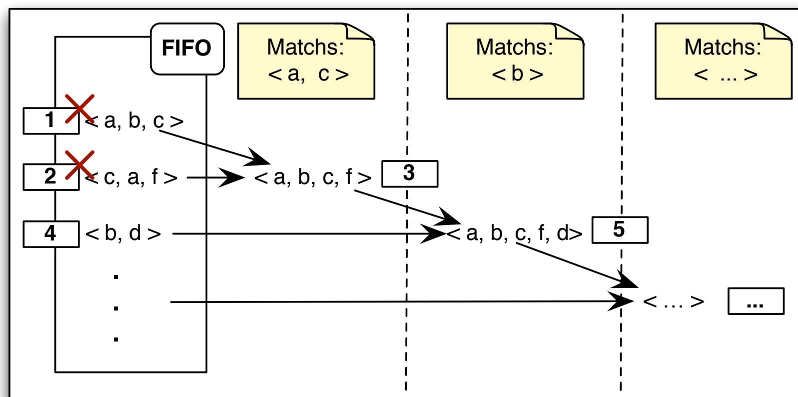


Figure 4.21 Algorithm used to find all possible matches

The order of the join nodes in the chain, and all possible matches has to be found before the execution, by exploiting the triple patterns that are composing the BGP operator. For this reason, we designed a simple algorithm that looks for all possible matches, into the triples, and in this way defines all Join node connections. Different join order can change the efficiency of the execution of the streaming application, however the order is not relevant for the correctness of the results. Since we are not interested in realising the most performant system, we not consider the problems of the joins order. For this reason our algorithm is not designed to provide the best solution, but just for giving as result a possible one.

Single Operators

Single nodes have correspondence one-to-one with the SPL nodes so they can easily translate in nodes of a streaming application. SPL primitive operators, can usually translate all these kinds of SSE operators with few exceptions, providing an implementation of their functionalities.

We can further subdivide the SSE operators, based on the number of input streams that they may require, in:

- *unary operators*: *CSV bgp*, *projection, extend, filters, order, group,* and *slice* SSE operators.
- *binary operators*: *join*, *leftjoin*, and *union* SSE operators.

In almost all example listed below we will reuse, for sake of clearness, the same example explained in the previous section (Section 4.1).

The **CSV bgp** is the SSE special Base Graph Pattern that, as we have said, cannot be considered like the other ones (Stream and Standard BGPs), because it presents a special configuration of its triple patterns. Since every triple pattern can be considered a single variable definition, the entire CSV BGP can be thought as a single stream definition, as one can see in the example below.

Figure 4.22 Example of transformation from a BGP CSV to stream schema

In order to translate this operator we need a node able to select the correct attributes of the stream and fill the CVS stream definition. That can be done, by exploiting the Functor SPL node, as one can see in the example in Figure 4.23.

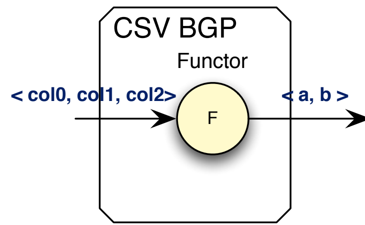


Figure 4.23 Example of CSV BGP single operator

The **project** and **extend** operators are SSE operators able to rename the input elements, producing some modifications into the data, and projecting the specified elements into the output. All these functionalities can be performed with the primitive Functor SPL node, as shown in the example below.

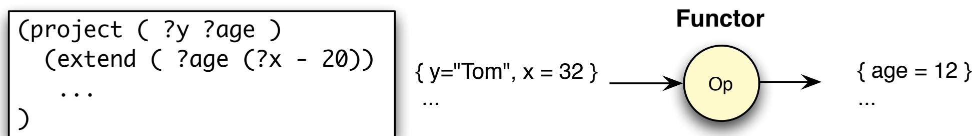


Figure 4.24 Example of project and extend single operator

The **filter** operator is the SSE operator that is responsible to filter the input elements, producing a subset of elements. The Filter SPL operator can be used to translate this operator, as one can see in the example below.

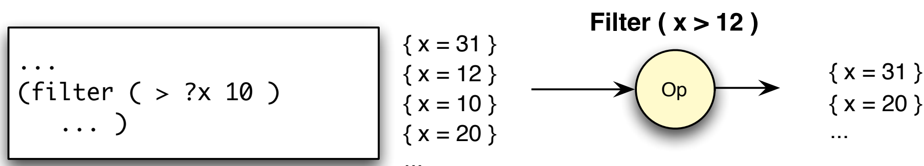


Figure 4.25 Example of filter single operator

The **order** operator is the SSE operator responsible for ordering input elements based on used specified constraints. The Sort SPL node can be used to translate this operator, as one can see in the example below.

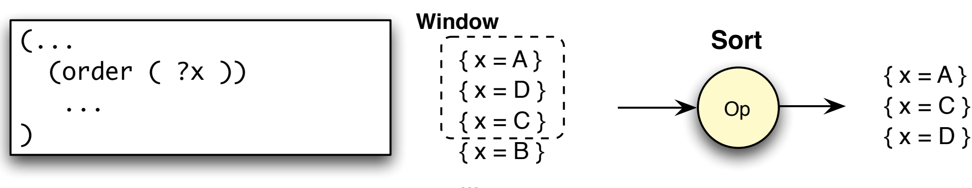


Figure 4.26 Example of sort single operator

The **group** operator is the SSE operator that is responsible for grouping input elements in different groups, and performing specific aggregate functionalities. The Aggregate SPL node can be used to translate this operator, as one can see in the example below.

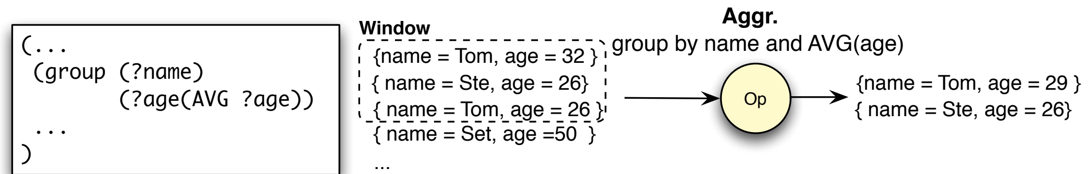


Figure 4.27 Example of group single operator

The **slice** operator is the SSE operator that is responsible for selecting a subset of elements from the results set. In this case SPL does not perform any specific node to implement this functionalities, so we configure a custom node in order to perform this action. The custom operator in this case just forward a number of user-specified results, by filtering all the other ones

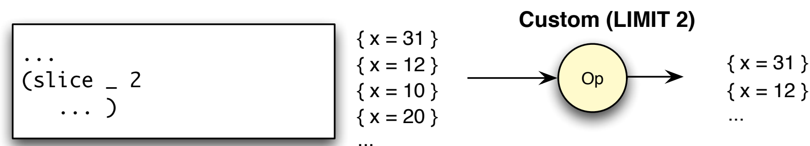


Figure 4.28 Example of slice single operator

The **join** and **leftjoin** operators are binaries SSE operators that are used to translate Group Graph Pattern and Optional Graph Patterns, and they are responsible for joining two Graph Patterns together. The Join SPL node can be used to perform both join, and left join, as we presented in the previous section (Section 4.1).

The input streams are taking from two nested nodes, so their results are joined together in order to produce a new single tuples into the output stream. In this case as in the BGP node, we exploit a simple algorithm to find all possible matches between the two input streams. An example of translation of a join SSE operator (corresponding Group Graph Pattern) is shown below.

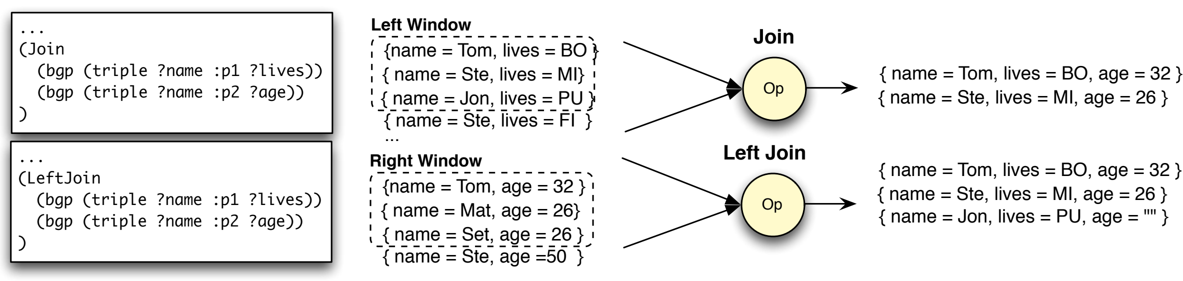


Figure 4.29 Example of Join/LeftJoin single operator translation

The **union** operator is the SSE binary operator that is used to perform the Alternative Graph Pattern. As we have previously said, we implement the union operator as Auxiliary node, even if in this case it is used to realise the logical union of two input streams.

4.2.2.3 Window configurations

Since we are working on data stream, we need to consider for all previous SPL nodes a proper window configuration, associated to every specific input stream by exploiting the *FROM STREAM* and *FROM CSV* input specifications. In this way, the defined window range and step that are used to configure all stateful SPL operators.

We have decided to support two different types of window configurations: *tuples-based* and *time-based* already presented in Section 2.3.1. The fist one is the simplest window that can be considered on a streams, based on the number of tuples that are selected within the window. On the other hand, the time-based window, provides two possible implementations, it can be based on:

- the *time system*: it is the easiest way to provide synchronisation in non-distributed environments, because the synchrony, between all different operators, is provided by the system itself. However this approach cannot be exploited in distributed execution environments, where the distributed machines cannot be easily shared the same system clock;
- the *time of data*: it is the best way to provide synchronisation in distributed environments, because in this way the timing of the distributed system is shared between all different machines with the data. In this way, all nodes of a streaming application, that can potentially performed on different machines, can be easily synchronised.

Even if the second approach represents the best solution, for this first implementation of our executor environment, we have chosen to work on a single machine, and for this reason we have exploited the time system.

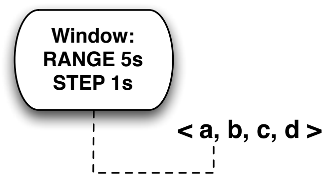


Figure 4.30 Example of Window configuration applied to a stream

The window specification can be considered as an additional attribute of stream specification, and for this reason it can be modified passing through different node of a streaming application.

Window operator modifiers

Even if it is possible to specify multiple and different range of window, in this first implementation we do not support different types of step specifications, so it is not possible define multiple windows with different steps. Starting from this assumption the possible modifications that can interest the window attribute of a stream are summarised by two classes of SPL nodes that we have specified:

- *fusion nodes*;
- *buffer nodes*;

Fusion nodes are those nodes that perform operations between two different input streams to provide an additional output stream with a new window configuration. For this reason, there are two possible cases, considering:

- *homogeneous input streams*: they are stream with the same window configurations;
- *heterogeneous window input streams*: they are stream with different window configurations.

The first case does not present any configuration problems, because the output stream will be configured with the same input window configuration. In the second situation it is possible to choose which between the two window configurations have to be used to configure the output one. The possible choices are either selecting the *biggest* or the *smallest* window configuration.

However, configuring the output stream with the biggest input window configuration, will lead incorrect window states during the execution of the streaming application. In this case, the tuples that are held in the smallest window configuration will remain into the window of the output stream even when they have expired. That situation can produce an inconsistent state of the window of the output stream, which keeps holding tuples that are no longer valid. Choosing to configure the output stream with the smallest input window configuration, will preserve the validity of the tuples contained into the window of the output stream, which in this case, will expire at the same time with the in all interested window.

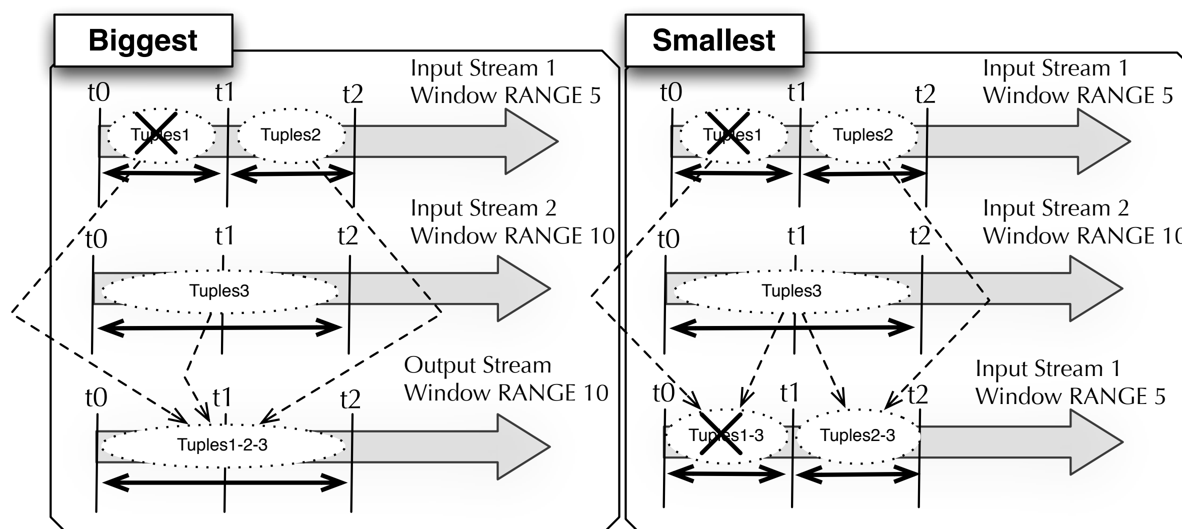


Figure 4.31 Biggest and Smallest output window configurations

In the above example (Figure 4.31), one can see both types of configurations: into the left part there is the configuration with the biggest window, and into the right part is contained the smallest window configuration. The example shows the different behaviours addressing when tuples 1 (a group of tuples into the window) expire. On the left part they remain into output window even after they are no longer valid tuples of the window of the stream 1. On the other hand, into the right part, they expire in the same moment with the window of stream 1.

The Join and the Union node are members of this class of operators, for this reason they have to be configured by following the previous descripted specifications.

Buffer nodes are those stateful nodes that perform actions for grouping (Aggregate node) tuples contained in a window that can be configured with range and step (in order to force the premature output). In this case, these operators are saving portions

of the stream into their window, as buffers, they performing their functionalities and they are sending out all results tuples. However, because they can be configured with a step parameter, it is possible to obtain all results of the node before that the entire window will be complete. In this way, every specified step the node will send out all contained values. This means that, if the buffer node is configured with a step ten times smaller than the range value, the results will be pushed out ten times before the window complete. In this way, the cascade node will receive the same tuples an unpredictable number of times during the same window range.

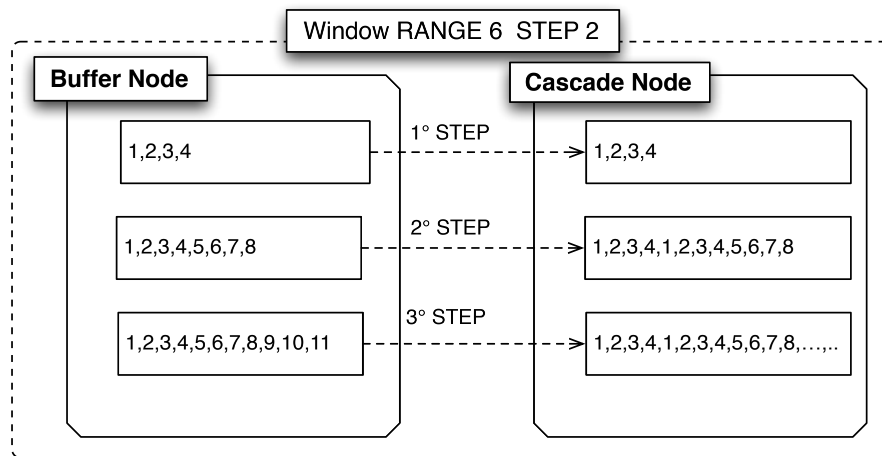


Figure 4.32 Example of problem lead by the buffer nodes

The above figure gives a clear idea of the problem, from the Buffer Node at every step of the window are sent out all tuples collected until that moment from the stream. For this reason, a generic cascade node will receive every time all results for every step.

This problem can be resolved by modifying the window configuration of the output stream by reducing the range value of the output window to the step value. In this way, the cascade node will perform its functionalities on a set of distinct tuples every time, because at every step the window will be complete and all items will be discarded.

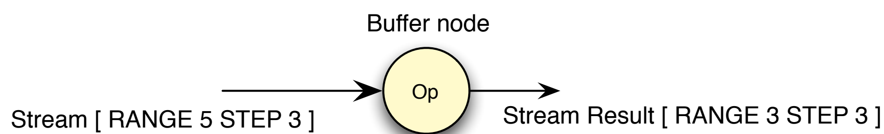


Figure 4.33 Example of Window configuration in a Bugger node

Window configurations in a streaming application

Considering both time and tuple-based window specifications we can identify two configuration levels:

- the *first level window configuration:* it is provided by the window specification obtains from the specific DubExtensions clauses, that interest all BGP nodes associated to the input. At this level, all contained stateful nodes are configured with homogeneous window specifications, and for this reason there is no problems into the window configuration;
- the *second level window configuration:* it is exploited by nodes that one can find in cascade of BGP nodes. At this level, the window configuration is obtained by the window configuration of node placed before of the node to configure, in this case, the window configuration are affected by the problems that we have presented.

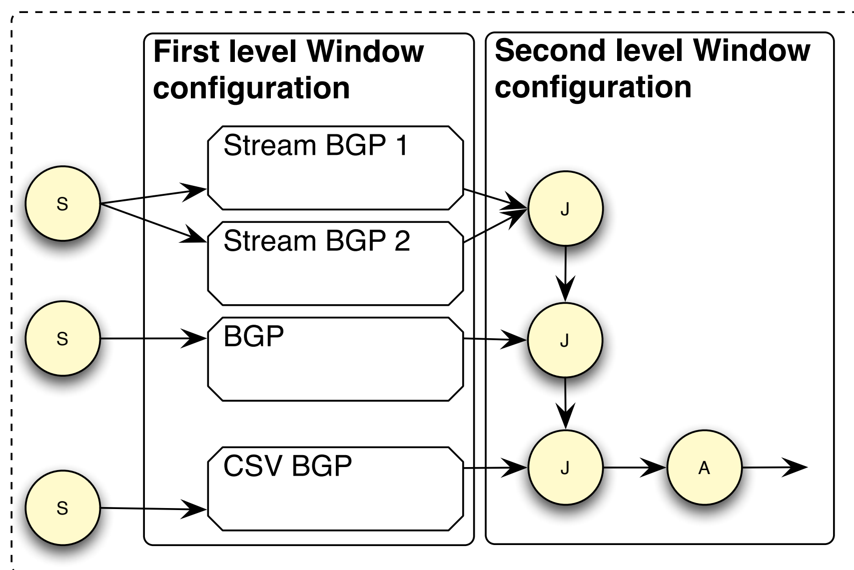


Figure 4.34 Two level window configurations

4.2.3 A complete translation example

Now that we have defined all the translation “rules” used to defined a streaming application from a DubExtensions query, in this section we want to give a complete example in order to provide a clearly idea of want is the result of the translation.

```
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://.../rfid>.
13      ?bikecount ibm:csvCol_2 <http://.../rfid>.}
14   } GROUP BY ?station ?stationid ?stationlat ?stationlong ?address
```

Figure 4.35 Example of DubExtensions query

We consider again the DubExtensions query that we have presented in the previous chapter (Chapter 3) and that we report again in the above figure (Figure 4.35).
As we have seen, this query pass thought a first level of translation that performs a first level of simplification in order to reduce the gap between the two languages DubExtensions and SPL. In this first level of translation, ARQ parser and analyser modules are exploited in order to provide and intermediate representation of the query with SPARQL S-Expression, as shown below.

```
1  (project (?station ?stationid ?stationlat ?stationlong ?address ?bike)
2      (extend ((?bike ?.0))
3        (group (?station ?stationid ?stationlat ?stationlong ?address)
4                ((?.0 (avg (/ ?bikecount ?numBike))))
5          (join
6            (bgp
7              (triple ?station dpPedia:agencyStationCode ?stationid)
8              (triple ?station dpPedia:maxNumOfBike ?numBike)
9              (triple ?station wsg84:long ?stationlong)
10             (triple ?station wsg84:lat ?stationlat)
11             (triple ?station address:streetAddress ?address)
12           )
13           (csv_bikestream:bgp
14             (triple ?stationid ibm:csvCol_0 <http://.../csv>)
15             (triple ?bikecount ibm:csvCol_2 <http://.../csv>)
16           ))))))
```

Figure 4.36 Example of DubExtensions query, after the first translation phase

Finally from this specification we have defined a set of transformation rules in order to provide a complete SPL streaming application definition, as shown below.

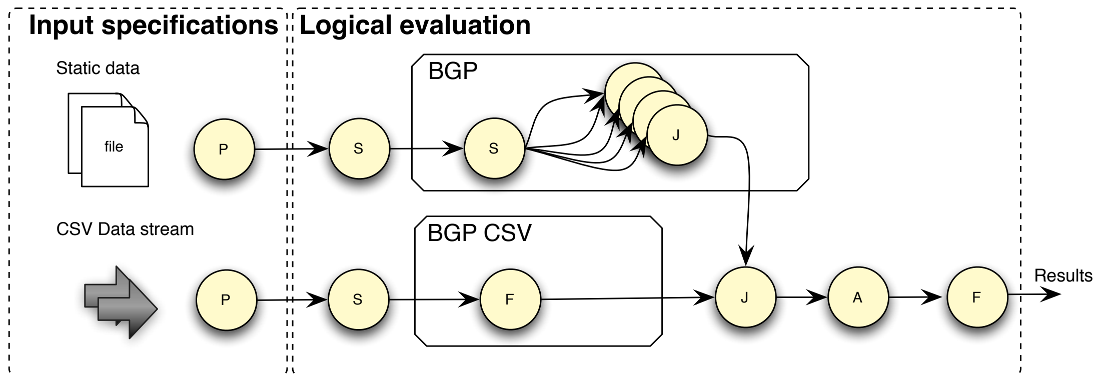


Figure 4.37 Example of DubExtensions query, after the second translation phase

Figure 4.37 shows an example of the input specifications that contain two types of input: static RDF and CSV stream. Two different parsers (*P*) process this information that is forwarded to the respective Split nodes (*S*), which are responsible to feed the two BGPs (BGP and BGP CSV). The output of the BGPs is joined with a Join node (*J*), and the node Aggregate (*A*), which calculates the aggregate functionality of average, manages its output. The last node is a Functor node that performs the projection of query.

We are now able to produce DubExtensions continuous queries that can continuously execute on InfoSphere Streams; moreover, we can exploit its potentialities for processing data streams and its skills for configuring streaming application to realise streaming applications that can be able to monitor smart cities.

5 BACKWARD STREAM REASONING

This chapter presents the implementation of backward stream reasoning approach, which provides a way to realise backward reasoning on needed tuples in order to answer continuous queries. In this way, our DubExtensions extension is able to provide more expressive queries able to completely exploit the advantage of working on structured information. For this reason, we have provided an additional module, a *backward stream reasoning module,* that provides all materialisations necessary to our streaming applications to answer complex queries. To reach this goal we have previously realised a forward static reasoner, even if multiple implementations are possible (the same Jena framework is able to reason on Linked Data), we want to investigate all possible strategies, to minimise the number of tuples that need to be computed, to improve the final performance of the system. For this reason, we have studied the dependencies between the RDFS rules, reducing the number of rules that could trigger other rules with their conclusions (Section 5.1), and optimising the reasoning approach by exploiting techniques of backward reasoning (Section 5.1.2). Moreover, after this analysis we present an implementation of both static and stream reasoner, by exploiting SPL nodes and defining a new module that could be added to SPL streaming applications (Sections 5.4, 5.2, and 5.3).

5.1 RULES DEPENDENCIES

In this first implementation of stream reasoning, we have exploited the set of basic rules, the thirteen RDFS Entailment rules (Section 2.1.3), since they represent the simplest set of rules, useful for reasoning on RDFS ontologies, and suitable for a first prototype implementation.

We have already presented the RDFS rules (Table 2.3), and we also have defined a subset of them, more "useful", composed by rules 2, 3, 5, 7, 9, 11, 12, and 13, that we will refer, from now on, in this analysis of rules dependencies. As we have said, the dependency relationships between rules are based on their *assumptions* and *conclusions*, and particularly on the fact that conclusions can also be assumption for

other rules producing a chain of activation, represented in a graph in the figure below (Figure 5.1).

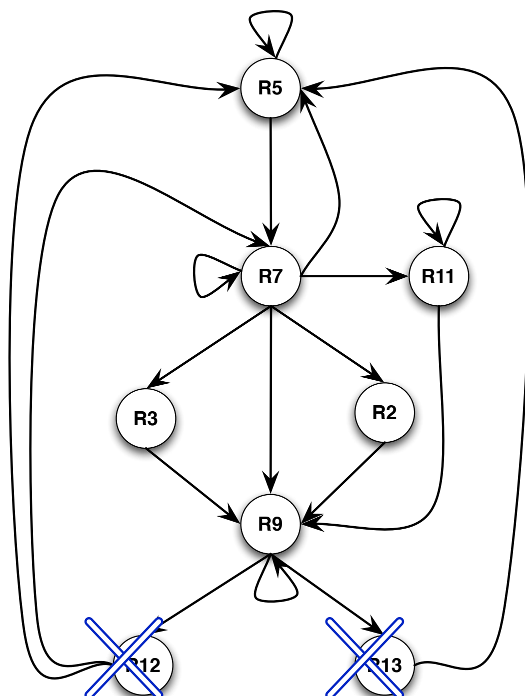


Figure 5.1 Dependency graph of relationships between the subset of RDFS rules **[30]**

Our first analysis concern rules 12 and 13, which as one can see in Figure 5.1, can be discarded from the subset of necessary rules [30]. Because, even if rules 12 is able to fire rules 5 and 7, and for this reason should be considered (we will use the notations of Table 2.3, referring to P_1 ,P_2, and M), if we are looking carefully to the single dependency, to activate rules 5 there must be either a sub-property P_1 of "rdfs:`member`" (P_2) or a sub-property P_1 of the property M. While to activate the rule 7, there must be some resources connected by the property in the conclusion of rule 12 (M). We do not consider the first case of rule 5, following the advice against Ontology Hijacking that we have presented in Section 2.1.2.3 of this dissertation. We decide to discard the other two cases, because they are not common, and in our use cases they do not appear and moreover, because they have been avoided in previous approaches on stream reasoning [30]. For the same reasons of rule 12, also rule 13 can be avoided because, even if it is able to fire rules 5 and 9, the necessary conditions for that to happen are really rare [30].
In this way, the left subset of RDFS rules is composed by rules 2, 3, 5, 7, 9, and 11, that are displayed in Table 5.1, which we will refer from now on.

	Subset of RDFS Entailment Rules		
	Assumptions	➔	*Conclusions*
2.	P rdfs:domain V & S P O	➔	S rdf:type V
3.	P rdfs:range V & S P O	➔	O rdf:type V
5.	P_1 rdfs:subPropertyOf P_2 & P_2 rdfs:subPropertyOf P_3	➔	P_1 rdfs:subPropertyOf P_3
7.	P_1 rdfs:subPropertyOf P_2 & S P_1 O	➔	S P_2 O
9.	C_1 rdfs:subClassOf C_2 & S rdf:type C_1	➔	S rdf:type C_2
11.	C_1 rdfs:subClassOf C_2 & C_2 rdfs:subClassOf C_3	➔	C_1 rdfs:subClassOf C_3

Table 5.1 Subset of RDFS Entailment rules

The above table represents the final subset of RDFS rules that can be used to reason over the majority of possible RDFS ontologies without losing results in a complete materialization, and it will be further analysed in the next section of this chapter.

The figure below shows the dependency graph between the new subset of rules, where we have highlighted the connections that present a single dependency between rules. For example rule 11 is able to trigger rule 9, but only providing the "C_1 rdfs:subClassOf C_2" assumption.

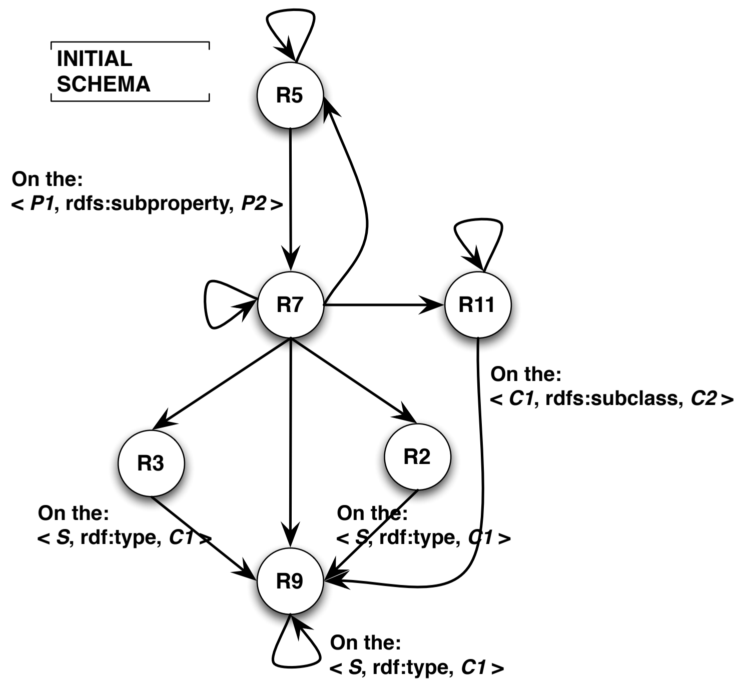


Figure 5.2 Dependency graph with explicit relationships

5.1.1 Triple pattern combinations

Starting from the dependency graph at Figure 5.2 our goals is to reduce the number of dependencies between all rules of the subset (Table 5.1), by considering all possible triple patterns that may be required in both SPARQL and DubExtensions queries. For this reason, we have computed all possible triple patterns that could be used in a query and we have realised a table of triple pattern combinations, shown below (Figure 5.3), where the upper case letters (X and Y) represent variables and the lower case letters (a, b, and c) represent generic constant values. It is important to note, that a triple pattern with either variables or constant values in two different points of the triples will be expressed with a combination where either the same upper case letters or lower case letters will be repeated. For example in the combination "`<X,b,b>`", the repeated "`b`" represents some predicate and object.

Combinations		
< X, Type, Y >	< X, SubClassOf, Y >	< X, SubPropertyOf, Y >
< X, Type, c >	< X, SubClassOf, c >	< X, SubPropertyOf, c >
< a, Type, Y >	< a, SubClassOf, Y >	< a, SubPropertyOf, Y >
< a, Type, c >	< a, SubClassOf, c >	< a, SubPropertyOf, c >
< a, X, Y >	< X, b, Y >	< a, X, c >
< X, Y, c >	< X, b, b >	< X, X, c >
< a, Y, Y >	< b, b, Y >	< a, b, c >

Figure 5.3 Combinations of all possible triples patterns

The table does not consider all possible combinations: some of them have been discarded because not possible ("`<a,a,a>`"), and other because do not provide any conditions useful to simplify the dependencies graph ("`<X,Y,Z>`"). Moreover, in the first phase of the analysis, we assume that the query pattern of a SAPRQL query is composed by only one triple pattern, in order to simplify the problem, thus we can study singularly how every possible combination can affects the dependency graph of the rules (Figure 5.2).

Each triple pattern represents a "schema" of all RDF triples that can be materialised from the input data, because it may be useful to answer a SPARQL query. For example, considering a SPARQL query that selects all possible subclasses of "`:Person`" (Figure 5.4), we will need to materialised all elements that correspond to the triple pattern "`<X,Subclass,c>`", as one can see in the example below.

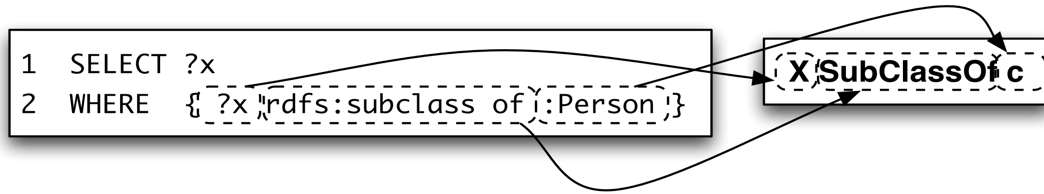


Figure 5.4 Correspondence triple pattern with triple combination

In this way, we can exploit this additional constraint in order to further reduce the number of rules needed to materialise all necessary information in order to answer the query. If we consider the conclusions of the rules (Table 5.1), one can see that not all rules can produce results that can interest a specific query, however they cannot be discarded yet; they may be useful in order to provide intermediate results that could trigger other rules necessary to satisfy the triple pattern conditions.

Considering again the previous example, the condition that needs to be satisfied is: "<X,Subclass,c>", which can be only produced by the conclusion of rule 11. However, as we have said, all other rules may be necessary in order to provide some conclusions that may trigger rule 11. For this reason, we have to check the dependencies graph (Figure 5.2), where one can see that the rules able to trigger rule 11 are rules 5 and 7.

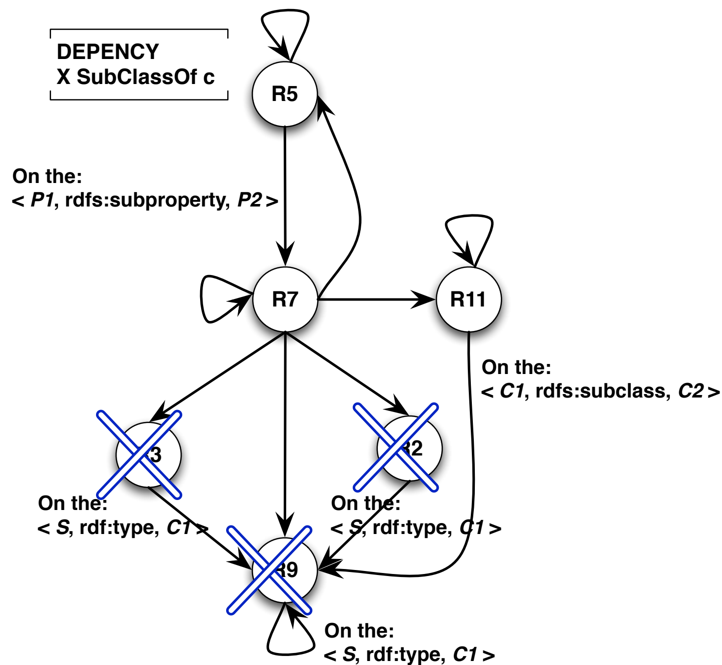


Figure 5.5 Dependency graph of the triple pattern combination *X SubClassOf c*

This means that all other rules can be not considered to materialise useful information to answer this specific query, as one can see in Figure 5.5.

A similar simplification can be provided for all possible triple patterns combination (Table 5.1) so that for every one of them can be found a specific subset of only necessary rules. All rules subsets obtained by every triple pattern combination are summarised in the table below (Table 5.2).

Combinations	R2	R3	R5	R9	R7	R11
< X, Type, Y >	✔	✔	✔	✔	✔	✔
< X, Type, c >	✔	✔	✔	✔	✔	✔
< a, Type, Y >	✔	✔	✔	✔	✔	✔
< a, Type, c >	✔	✔	✔	✔	✔	✔
Combinations	**R2**	**R3**	**R5**	**R9**	**R7**	**R11**
< X, SubClassOf, Y >	✖	✖	✔	✖	✔	✔
< X, SubClassOf, c >	✖	✖	✔	✖	✔	✔
< a, SubClassOf, Y >	✖	✖	✔	✖	✔	✔
< 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 >	✔	✔	✔	✔	✔	✔
< X, Y, c >	✔	✔	✔	✔	✔	✔
< a, Y, Y >	✔	✔	✔	✔	✔	✔
< a, X, c >	✔	✔	✔	✔	✔	✔
< X, X, c >	✖	✖	✔	✖	✔	✖

Table 5.2 Summary table of subset of rules for every combination

As a summary, it is possible to see that for the triple pattern combinations with:

- the *Type* predicate, the subset of rules cannot be further reduce;
- the *SubClassOf,* and *SubPropertyOf* predicates, the subset of rules are reduced to rule 5, 7 and 11;

- the constant (*b*) predicate, which represents all possible predicates with the exception of *Type, SubClassOf,* and *SubPropertyOf*, the only necessary rules are rules 5 and 7;
- the variable (*X* or *Y*) predicate, different from both subject and object, the subset of rules cannot be further reduced;
- the variable (*X*) predicate, equals to the subject, the only necessary rules are rules 5 and 7.

Moreover it is interesting to state that rules 5 and 7 are subset of rules in which it is not possible to apply filters and that are always needed in every classification. That happens because, rule 7 is able to produce generic triples ("$\langle S, P_2, O \rangle$"), and because rule 5 is recursively connected to rule 7.

At the end of this analysis, we have obtained an interesting simplification of the set of rules needed to perform reasoning on every single triple pattern combination. In this way, we have reduced the number of tuples necessary to compute every complete materialisation, by a forward reasoner. However we want to further reduce the number of triples needed, by exploiting the backward reasoning techniques of the next section.

5.1.2 Backward reasoning filters

Starting from previous results, we want to further reduce the number of triples for our reasoner, by exploiting backward reasoning techniques. The backward reasoning realises the reasoning just on the minimal number of triples necessary to compute the required materialisation in order to reach a specific goal. Therefore, we can exploit the specific subset of rules, defined for every single triple pattern combination, and then we apply a specific set of filters to them. Any filter does not have to eliminate any useful results, even if it has to be as strict as possible in order to significantly reduce the number of computed triples. We have realised a backward chaining graph considering all possible triple pattern combinations as goals of the graph and looking for all possible derivations obtained by exploiting its specific subset of inference rules. For every obtained derivation, we can identify the filter that could be applied to every assumption of the derived rules without loosing results, as shown below.

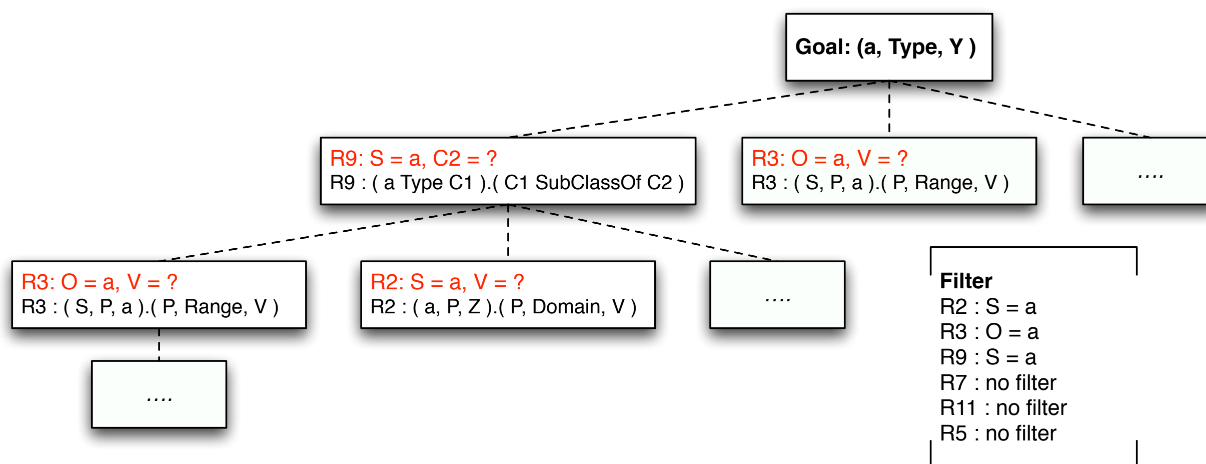


Figure 5.6 Backward chaining on "*a Type Y*" combination

The Figure 5.6 shows an example of backward chaining, realised on the specific combination "<a,Type,Y>", and the resulting set of filter for this specific goal. By applying backward chaining on all other possible triple pattern combinations, we can obtain all possible filters for every one of them, summarised on the table below.

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 >	✖	✖	✔	✖	✔	✖

Table 5.3 Summary table of all possible filters applicable to all possible triple combinations

The filters represent constraint conditions on the resources and properties on the assumptions of the rules, which reduce the number of triples that every single rule can accept and need to calculate. For example, considering the triple pattern of before: "<a,Type,Y>" and the filter applied on rule 2 (S = *a*), this means that a triple in order to be a valid assumption of rule 2, need to provide a subject (*S*) that have to correspond to "*a*" (where "*a*" is the value of a triple template). For example considering the triple pattern ":Student1 rdf:type ?x", we can say that the only necessary conclusion that could be generated from rule 2 in order to answer the query are generated from assumptions where *S* is identified with ":Student1" (S = :Student1). Therefore, in this case rule 2 can be re-defined as shown below.

```
2.  P rdfs:domain V
    :Student1 P O      &   ➔   :Student1 rdf:type V
```

By exploiting all defined filters of Table 5.3, we can implement backward reasoning strategies for every single triple pattern of a SPARQL query. However, usually a SPARQL query provides more triple patterns for every query pattern, depending on the complexity of the query: so we need to deal with all of them. We could realise a backward reasoning on every single triple pattern separately: in this case, the reasoner needs to calculate the materialization of every single triple pattern, even if they lead the same materializations, with obvious performance problems.

Indeed in this way, the same reasoner could calculate the same conclusions several times, while in a better approach, all triple pattern could be computed simultaneously, so they can share the same intermediate results. Starting from this assumption, if a query contains multiple triple patterns we decide to "merge" the filters of every single triple pattern combination (Table 5.3), providing the less strictly constrained filter configuration on the rules assumptions. So, we are able to calculate all materialisations needed to all triples patterns simultaneously and sharing the intermediate results.

< a, Type, Y >	**S = a**	**O = a**	**✔**	**S = a**	**✔**	**✔**
			+			
< X, SubClassOf, Y >	**✖**	**✖**	**✔**	**✖**	**✔**	**✔**
			=			
< a, Type, Y > **< X, SubClassOf, Y >**	**S = a**	**O = a**	**✔**	**S = a**	**✔**	**✔**

Figure 5.7 Example of merge of two triple pattern combinations

If we consider for example a query pattern composed by ":Student1 rdf:type ?x" and "?x rdfs:subclassOf ?y", the triple pattern combinations are respectively "<a Type Y>" and "<X SubClassOf Y>". Therefore considering all set of filter applied to every single combinations, the final filter configuration is obtained by merging every single condition that are exploited by the two combinations, as in Figure 5.7.

Note that, the final filter produced will be less constrained, to not loose any results, and in this way, based on the number of triple patterns contained in a SPARQL query, our backward reasoner could be programmed every time with less filters, until that, there are no constraints remained, to be configured. In this case our backward reasoner will perform like a forward reasoner.

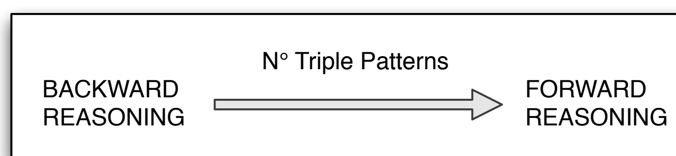


Figure 5.8 Relationships between Backward and Forward Reasoning

However starting from this assumption, we can say that the backward reasoning approach is always better performing if compared to the same forward reasoning implementation, since this last one represents its worst filter configuration case.

5.2 STATIC BACKWARD REASONER

This section presents the design of the implementation of a static backward reasoner, in order to realise an independent module capable of computing materialisations on static RDF triples. To insert this reasoner in our streaming application, we have to exploit again SPL nodes, in order to connect this new module to the previously defined streaming applications (Chapter 4).

In order to implement this module we have a set of SPL nodes composed by:

- the Split and Union nodes (that we have defined), responsible to manage the routing of the triples (tuples);
- the Join SPL node responsible to implement the subset of rules previous defined (Table 5.1);

- the Filter SPL node responsible to select the correct results from the materialised tuples.

In this case, because we are working in a static environment we do not need of window configuration, and all join nodes can be configured with an infinite window (in this way, the contained tuples will never expire).

Next sections present step by step all fundamental steps to compose the static backward reasoning module.

5.2.1 The rules connections

Every rule can be translated as a Join node, specifying the correct input tuples as assumptions and a correct output triple as conclusion, as in the example below.

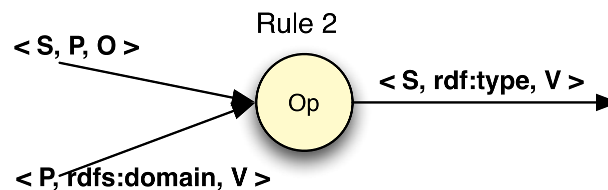


Figure 5.9 Example of Join node configuration to implement Rule2

Moreover, all materialised conclusions are unified to form the *intermediate results* of the reasoning, to be used to trigger other rules. However, not every rule is connected to all other ones, because their connections are commanded by the dependencies graph, that we have presented in the previous section (Figure 5.2).

In order to route every tuple to the correct Join node we exploit a Split node, able to manage multiple different input/output streams.

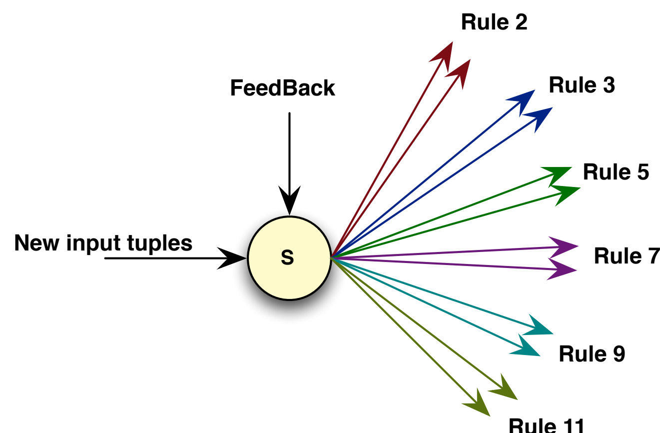


Figure 5.10 Split node configuration in the static reasoning module

In this way, the Split node is responsible for routing the input tuples, based on they predicate values, managing the rules connections of the reasoning module. At the same time for the routing of the tuples, the Split node can also be configured to perform filtering actions, by forwarding just those tuples that respect the specified constraints. In this way we can use the Split node also to perform the set of backward filter expressed for every rule (as specified in the previous section).

5.2.2 Output filters configurations

In order to manage the output tuples from the reasoning module, we have exploited two different Filter nodes. Both filters are used to apply specific constraints on the output tuples by filtering:

- tuples that are coming directly from the input streams, since they may be necessary to directly answer the query, without any need of passing though the reasoning module;
- tuples that have been materialised by the subset of rules, and that compose the *intermediate results* of the reasoning module; they do not have to be considered as results of the reasoning module, because not necessary to answer a specific query.

Specific triple pattern combinations that compose the query pattern configure both Filter nodes. For example considering again the query: select all types of *Person*, the only triple pattern necessary to answer this query are those ones that satisfy the condition: “`<X Type :Person>`” (“`<?x rdf:type :Person>`”), and they represent the only combination needed to answer the query.

5.2.3 Complete static backward reasoning module

Considering the whole previous sections, we can now provide a complete architecture overview of the reasoning module that, based on the specific filter configurations, can perform backward and forward reasoning together. Figure 5.11 shows an example of representation of the structure provided by the SPL nodes in order to realise reasoning on tuples of Linked Data.

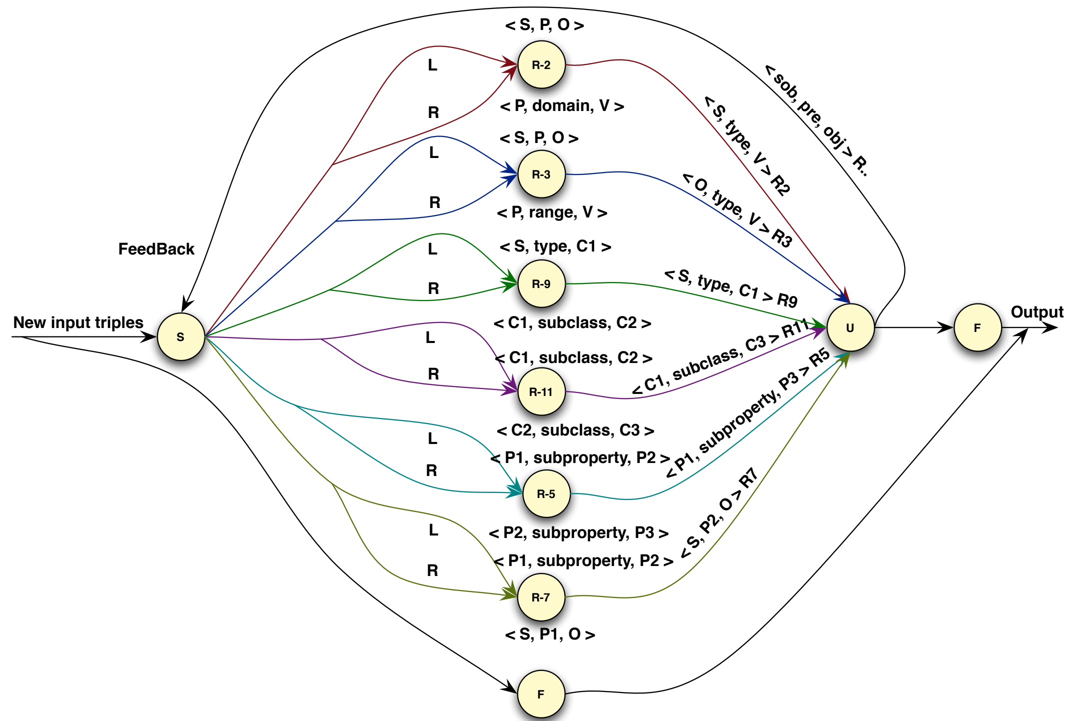


Figure 5.11 Logical representation of the static stream reasoning module

The Split node (*S*) is responsible for sending the correct tuple to the correct rules (R-2, 3, 5, 7, 9, 11), and the figure explains both assumptions and conclusions of every single rule. The Union node (U) unifies all conclusions materialised by the rules, and they are sent back to the Split node. The Split node forwards the feedback conclusions again to the correct rules, according to which rule has generated them, and by exploiting the dependency graph.

The final output of the reasoning module is composed by all tuples that pass through the Join node, which implement the rules, and by those ones that are coming from Filter node (F), applied directly in the input tuples.

5.3 STREAM BACKWARD REASONING

In the previous implementation we assume to work on static RDF data; however our goal is to produce the stream backward reasoning module to deal with RDF stream information. These data can be used to describe real-time environments, even if the information can be both static (Knowledge) and real-time (Data). The Knowledge represents the static and the schema information used to describe the Data, the real-time information. For this reason, even if a stream reasoning module has to deal with

both type of information, it is possible to exploit a standard static reasoner in order to reasoning on the Knowledge. Starting from these assumptions, the stream reasoning module can be subdivided in two different components:

- *static component*: generic static reasoner, whose duty is to provide reasoning on static schema information (in our case we will use the reasoner that we have defined);
- *streaming component:* responsible of performing reasoning on streams.

The two components need to be connected in order to provide reasoning on data streams, since the static component has to deal with the Knowledge, and the streaming component with the Data. In this way, the static component realises the static backward reasoning on schema information, which are used as assumptions from the streaming component.

	Streaming component rules			
	Assumptions		➔	*Conclusions*
2.	`P rdfs:domain V` `S P O`	&	➔	`S rdf:type V`
3.	`P rdfs:range V` `S P O`	&	➔	`O rdf:type V`
7.	P_1 `rdfs:subPropertyOf` P_2 `S` P_1 `O`	&	➔	`S` P_2 `O`
9.	C_1 `rdfs:subClassOf` C_2 `S rdf:type` C_1	&	➔	`S rdf:type` C_2

Table 5.4 The subset of rules exploit by the streaming component

Since the stream contains only Data there is no need to exploit rules that can only produce schema information that are static by definition. For this reason rules 11 and rule 5 can be eliminated from the subset of rules exploit by the streaming component. All rules left (Table 5.4) are necessary and required to work on stream of information, because they can to infer instances of solutions. However, in order to archive conclusions, they should consider both assumption coming from the Knowledge and from the Data, since part is expressed by schema information ("`P rdfs:domain V`", "`P rdfs:range V`", "P_1 `rdfs:subPropertyOf` P_2", and "C_1 `rdfs:subClassOf` C_2"), and part comes from both static and stream data. For this reason, we have to consider that the Join nodes, used to implement these set of rules have to be loaded by both static and stream information. This means that one side of the window of the join we be loaded with static triples, the other side with stream

information. The static information are all materialised by the static backward reasoner, which is the responsible for feeding the streaming component, which has to use the static information, in order to deal with the stream data and to produce the materialisations of the conclusions of the rules.

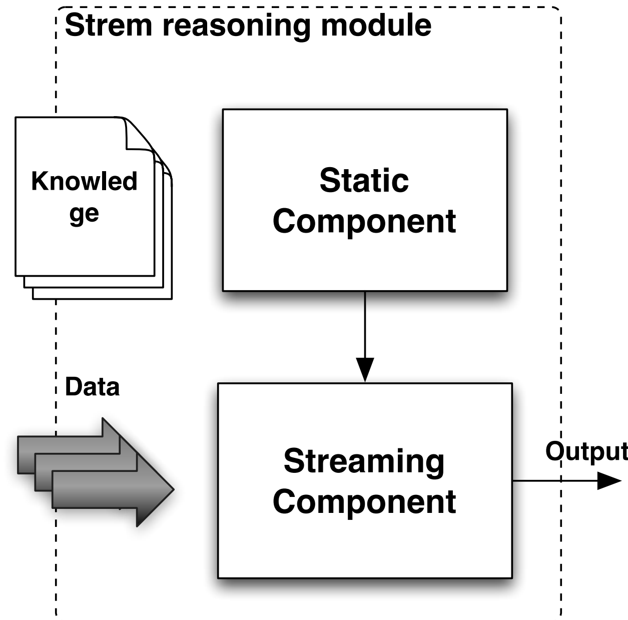
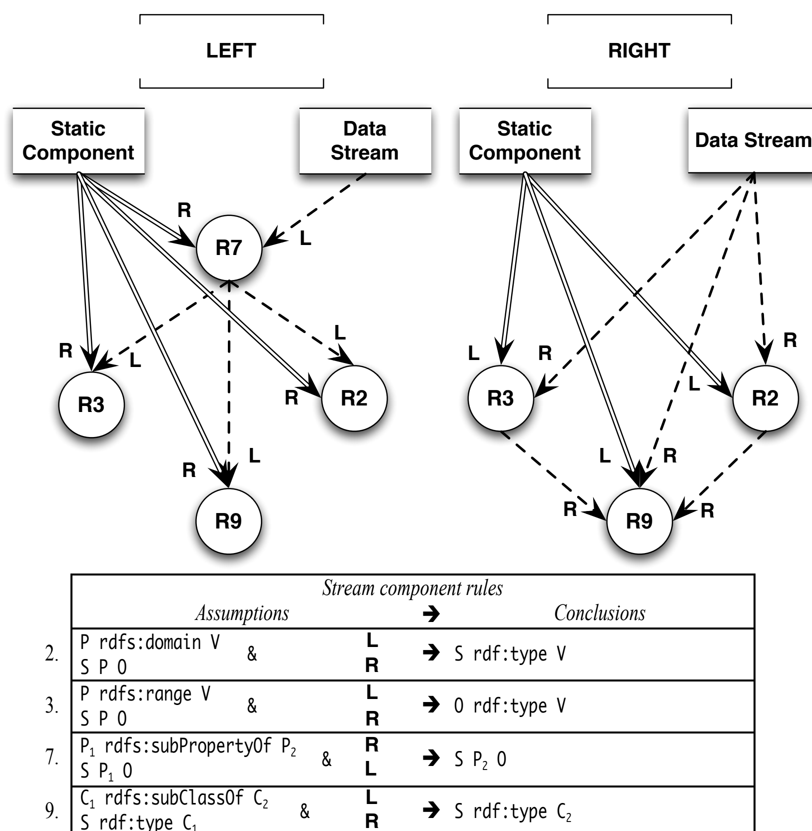


Figure 5.12 Overview of the Stream reasoning module

5.3.1 Streaming component

The Streaming component is the only part of the Stream reasoning module that has to deal with stream information, and it exploits schema information materialised by the Static component, since it can produce useful assumptions to infer new conclusions from the stream. For this reason, the Streaming component presents different types of inputs with two window configurations: either with expiration condition (data stream) or that never expire (static data); both are necessary indispensable to work on data stream. Due the different expirations applied to every stream, they have to be kept separately; that means that we need to provide two different combinations of rules that are loaded by the two different types of inputs, as in the figure below.



	Stream component rules: Assumptions		→ Conclusions
2.	P rdfs:domain V & S P O	L R	→ S rdf:type V
3.	P rdfs:range V & S P O	L R	→ O rdf:type V
7.	P_1 rdfs:subPropertyOf P_2 & S P_1 O	R L	→ S P_2 O
9.	C_1 rdfs:subClassOf C_2 & S rdf:type C_1	L R	→ S rdf:type C_2

Figure 5.13 Configuration rules in the Streaming component

Figure 5.13 reports the assumptions of the rules divided in left (**L**) and right (**R**) assumptions, and they are used to realise the two different combinations of rules, called:

- *left*: where the Data Stream load the left part of the rules, that correspond to the left window of the Join node that implement the rule;
- *right*: where the Data Stream load the right part of the rules, that correspond to the right window of the Join node that implement the rule;

The static component is used to load the static materialised data into the opposite size of the window, where the data stream is responsible of configuring the dependencies between rules, since they are affected by the same window expiration condition (because the window configuration is stricter than the window for the static data).

As one can see, by considering the two different configurations, there are some missing rules and dependencies between the rules; this happens because not all assumptions can be found into the data stream and some connections can actually be

discarded. Considering the case of rules rule 7 (Figure 5.13), it does not appear in the *right* configuration because the right assumption of the rule (“P_1 rdfs:subPropertyOf P_2”) cannot be found on the stream, so the rules will never produce its conclusion, and it can be discarded. We have defined two new dependencies graph to be used in order to configure the forward stream reasoning module. However, to realise backward stream reasoning, we must consider the filter configurations. These filters, the same we have previously defined (Section5.1.2), exploiting a less extensive set of rules. In this way, we can further reduce the number of tuples, improving the performance of our backward reasoning approach.

5.3.2 Complete backward stream reasoning module

The backward stream reasoning module presents the same operators that we have exploited to produce the static one; however, in this case, the architecture of the streaming application is more complex, because it consists of different components, as shown below.

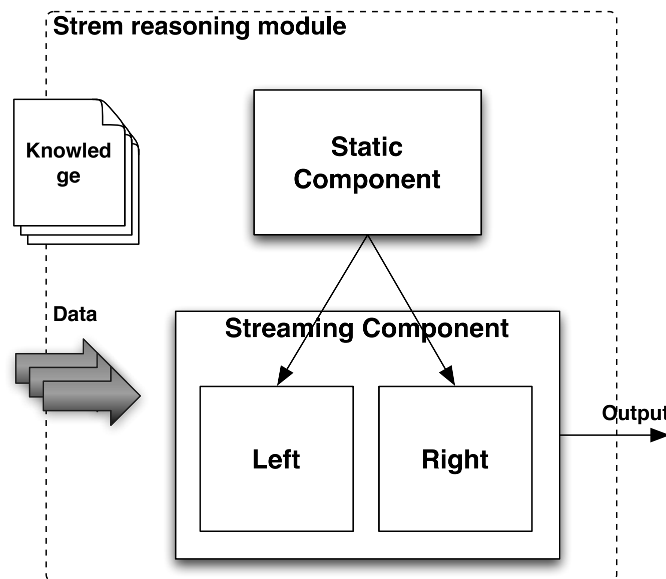


Figure 5.14 Logical Architecture overview of the Stream reasoning module

It is interesting to note that, the executions of the static and dynamic component cannot be at the same time. The static component must execute first, because it materialises the necessary information from the Knowledge to load the left and right parts of the streaming component. Only when all needed data have been generated and loaded into the Join node of the stream component, it can start executing.

The SPL implementation of the left and right parts of the streaming reasoning component are really similar to the static reasoning module, even if in this case the routing rules provided by the Split node are realised, with different dependencies graphs (Figure 5.13).

Moreover the same as before, the Split node is used to filter the incoming tuples, based on the filter configurations defined to realised backward reasoning.

In the following, we present how this module can be configured and connected to the streaming applications generated by DubExtensions continuous queries.

5.4 REASONING MODULES IN STREAMING APPLICATIONS

In the previous sections, we have defined the two types of reasoning modules exploited to materialised data from the incoming static and stream information. However in order to use the materialised information, we need to connect these modules to the stream application, as below.

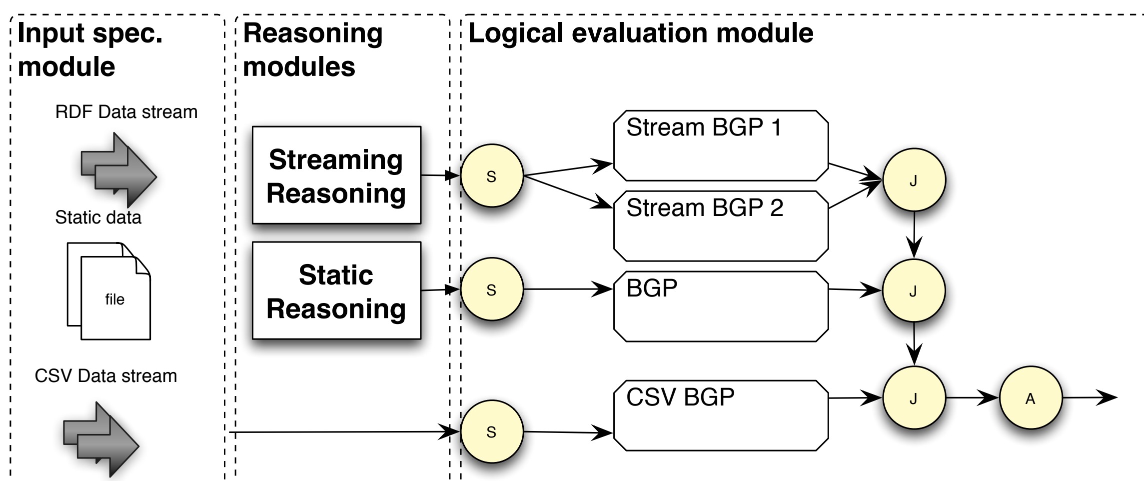


Figure 5.15 Complete overview of a Streaming application with reasoning modules

The two reasoning modules are organised between the *logical evaluation* and *input specifications module*, providing all input tuples that are necessary to answer a specific query, and executing in both forward and backward modes. Moreover, because it is not possible realise reasoning over unstructured data, there is no reasoning module for CSV streams.

Both reasoning modules are configured depending on the triple patterns contained into DubExtensions queries, in order to produce the strictly necessary tuples that are needed to answer the request. However, the streaming reasoning module needs an additional configuration of the window associated to the stream. Moreover, it also needs to provide static input specifications, to configure the input of the static component. These static inputs can be specified with the DubExtentions clause *FROM ONTOLOGY*, which specifies a static input to be shared between both static and stream reasoning modules.

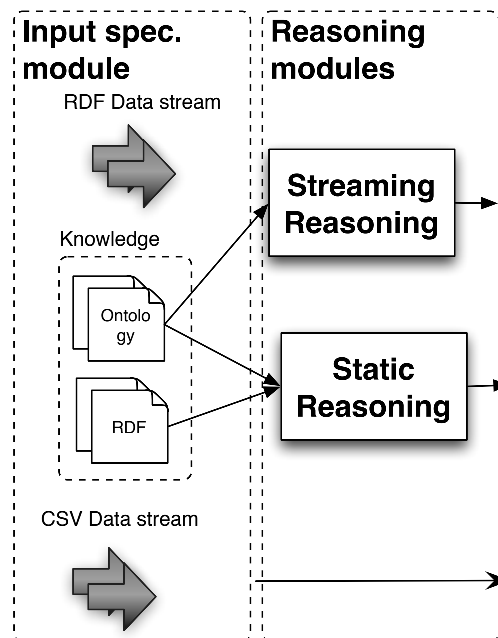


Figure 5.16 Functionality of FROM ONTOLOGY clause

The stream reasoning module can be configured with the specific window specification provided from the input stream. Therefore, for multiples window configurations, a different reasoning modules need to be configured for every one of them. Our first implementation allows only specifying homogeneous window configurations for the input streams.

This chapter has presented the last element to realise streaming applications, able to answer complex DubExtentions query, and executing on heterogeneous data streams. Next chapter of this dissertation (Chapter 6) presents a real use case: the city of Dublin, where we have employed our system, to execute on real-time information, provided in heterogeneous formats, and combining all these data with reasoning techniques.

6 REAL USE CASE AND PRELIMINARY PERFORMANCE

This chapter presents a real use case, where we exploit our approach to process heterogeneous data streams. Our use case describes and organises heterogeneous stream information coming from different sensors placed in Dublin city areas. Our goal is to exploit DubExtensions queries in order to generate streaming applications able to process the amount of heterogeneous data, coming from the city. In this way we are able to query a collection of data that, because their complexity, need to be evaluated with specific indicators, to extract the meaning of their measurements. For this reason in our approach we want to provide Key Performance Indicators (KPIs) to capture the evolution of quality of life as well as tracking abnormal situations in different areas of the city. More specifically, we have developed a DubExtensions query, whose aim is to identify interesting changes into the city, based on several different data sources (static and streaming), thus that it can be easily monitored, by identifying the most interesting areas that have produced the changes (Section 6.2). To realise this complex evaluation of a real-time environment, we also exploit our backward reasoning approach, in order to be able to reason over static and stream data. For this reason, before start by presenting the use case, the most interesting achievement of this project, we provide some preliminary performance evaluations of the backward stream reasoning approach (Section 6.1).

All tests and simulations have been realised on an IBM blade server with 12 CPUs Intel(R) Xeon(R) X5690 at 3.47GHz, 100GB of RAM, configured with Linux OS (Red Hat 4.1.2-52), and exploiting InfoSphere Streams 2.0.0.3.

6.1 PERFORMANCE EVALUATION

This section presents some preliminary performance evaluation of our backward stream reasoning approach and a further comparison between forward and backward stream reasoning implementations, by testing the abilities of the system of scaling well, over an increasing number of input triples and different window

configurations. To implement these evaluations, we have exploited *Lehigh University Benchmark (LUBM)* [50], used to evaluate queries over a large static dataset, by executing with a single realistic ontology, and designed to work in the specific university domain [50]. Since our system is an initial prototype that still needs to be developed, we cannot use other ad-hoc benchmark for stream reasoning evaluations, because we have not supported yet all necessary features. For this reason, we have customised the static dataset, provided by LUMB, providing a division of data in static and stream information. Since LUMB works in the university domain, we consider static information all data that refer to universities, professors, researchers, departments, and courses, while everything that concerns data of students have been considered as a stream. Moreover, LUMB benchmark also provides a set of queries, which can be used to provide different query patterns in order to test the different capabilities of reasoning approaches. Into the next subsection (Subsection 6.1.1) we will present these queries, while into the next one, we will provide their evaluations on data streams (Subsection 6.2.2).

6.1.1 LUMB queries

LUMB provides a set of 14 queries, that are used to test the different capabilities of a reasoning system, however due the customisation that we have realised to the dataset, we have defined a subset of these queries, excluding those that only perform on static information, because they are superficial to evaluate our stream reasoning approach. Therefore, we only exploit 9 queries of the entire set, since they are the only one that contain at list a triple pattern that refers to stream triples (the students).
The different queries provided by LUMB are used to test, capacities of reasoning systems for deal with [50]:

- *large input and high selectivity queries*: they provide a large set of triples as input, but a small set of triples as output, and their complexity is in finding a reductive set of triples in a large dataset;
- *large input and low selectivity queries*: they provide a large set of triples as input, and a large set of triples as output, and their complexity is in providing large amount of triples as output;
- *reasoning on implicit relationships*: those are relationships that need to be found from second level conclusions (with a conclusion inferred from another

conclusion) and for this reason they can be used to test the dependency relations of the inference rules of a system;

- *reasoning on explicit relationships*: relationships that can be found directly into the ontology definition, explicitly provided with a specific property: for example subClassOf;
- *several classes and properties*: the simultaneously presence of several classes and properties can provide and high level of complexity in order to answer the query, since the system needs to consider more information at the same time;
- *presence of triangular pattern*: they are the most complex pattern provided by LUMB query, which is realised between three triples patterns related in a triangular form of relations (*<?x advisor ?y> . <?x takeCourse ?z> . <?y teacherOf ?z>*);
- *wide hierarchies*: some concepts are provided with a large set of sub-classes and sub-properties that realise a deeply hierarchical organization of data. For example the *Person* concept, that it is the super-class of several other ones, needs more complex computations in order to be materialised, since deeply organised.

Every query can combine one or more of these requirements, providing different type of tests with different levels of complexities. The table below represents a summary of capabilities that are tested by every query and that we have exploited in our test of performance.

Query ID	**Requirements**
Query 1	Large input and high selectivity queries
Query 2	Several classes and properties Presence of triangular pattern
Query 5	Reasoning on implicit relationships Reasoning on explicit relationships Wide hierarchies
Query 6	Large input and low selectivity queries Reasoning on implicit relationships Reasoning on explicit relationships
Query 7	Large input and high selectivity queries Reasoning on implicit relationships Reasoning on explicit relationships Several classes and properties
Query 8	Large input and high selectivity queries Reasoning on implicit relationships Reasoning on explicit relationships Several classes and properties

Query 9	Large input and high selectivity queries Reasoning on implicit relationships Reasoning on explicit relationships Several classes and properties Presence of triangular pattern Wide hierarchies
Query 10	Large input and high selectivity queries Reasoning on implicit relationships Reasoning on explicit relationships Wide hierarchies
Query 14	Large input and low selectivity queries

Table 6.1 Subset of LUMB query and relative classification **[50]**

In order to be executed in our system, the subset of necessary queries, provided by LUMB, need to re-write in DubExtensions and translate in a SPL streaming application in order to be able to execute on InfoSphere Streams. In the example below, one can see an exemplificative translation of a LUMB query (query 2).

```
SELECT ?x ?y ?z
FROM ONTOLOGY </home/tallevis/SPLQuery/InputData/lubm.nt>
FROM </home/tallevis/SPLQuery/InputData/Static_University0.nt>
FROM STREAM </home/tallevis/SPLQuery/InputData/Dynamic_University0.nt> [RANGE TRIPLES 66774] AS 'students'
WHERE {
        {?y <http://...#type> <http://.../univ-bench.owl#University>.
         ?z <http://...#type> <http://../univ-bench.owl#Department>.
         ?z <http://.../univ-bench.owl#subOrganizationOf> ?y.}
      STREAM 'students' {
         ?x <http://...#type> <http://.../univ-bench.owl#GraduateStudent>.
         ?x <http://.../univ-bench.owl#memberOf> ?z.
         ?x <http://.../univ-bench.owl#undergraduateDegreeFrom> ?y}
      }
```

LUMB Query 2

Figure 6.1 LUMB Query 2 expressed in DubExtensions

Moreover, based on their translation we can also provide a further classification in:

- *stream/static queries*: those queries that need both stream and static information, in order to complete the computations required; they are the most complex queries and need to exploit the LUMB ontology and reasoning on stream and static information to be able to answer the query. This class comprehend query 2, 5, 7, 8, and 9, which, in order to deal with static and stream data, are translated with two different BGPs (Figure 6.1);
- *reasoning stream queries*: those queries that need to compute reasoning only on stream information in order to answer the query. This class comprehend query 6, 10, 14, which are translated with a single stream BGP, and that only exploit the stream reasoning module;

- *no reasoning stream query*: it is a query that does not need any kind of reasoning to answer the query, because that is obtained just selecting the tuples from the stream. This class comprehend only query 1, which is translated with a single stream BGP.

Note that, in order to test our system, we have exploited a tuple-based window specification (*TRIPLES*), because we need to know precisely the number of triples that have been selected and that are executing in our system. However, the selection provided by a tuple-based window configuration only affects the input triples, and not all materialised information that could appear into the reasoner. Moreover, since the ontology provided by LUMB is expressed in OWL, and our system is only able to realise reasoning on RDFS rules, the results provided by our reasoning system could be different from the effective results provided by LUMB. However, because we need to check if our system is performing correctly, we have exploited a widely tested system, Jena framework, to compute the expected results for every single query, considering just the subset of RDFS rules.

Benchmarck	LUBM Answers	Jena RDFS	Backwrd Stream Reasoner
Query 1	4	4	4
Query 2	0	0	0
Query 5	719	719	719
Query 6	7790	6463	6463
Query 7	67	61	61
Query 8	7790	6463	6463
Query 9	208	134	134
Query 10	4	0	0
Query 14	5916	5916	5916

Table 6.2 LUMB queries results with OWL rules and with RDFS rules

By exploiting the results of Jena, we have tested the correctness of our system, to reason with RDFS information, on a dataset generated by LUMB, by exploiting the configuration with a single university, that is usually identified by *LUBM(1,0)*, with a set of 100K triples.

6.1.2 Preliminary performance

By exploiting the LUMB benchmark and its set of queries, we want now to provide a comparison of forward and backward stream reasoning approaches. Thus we can prove if the backward filters combinations are useful, to reduce the input tuples, and if they are able to increase the performance of the system. After these firsts tests, we also focus our evaluation on testing the backward stream reasoning approach, and more exactly we will provide some considerations of its capabilities of scaling with increasing window sizes.

6.1.2.1 Forward/Backward Reasoning

During this first test we exploited a fixed tuple-based window configuration of 66774 triples, increasing the number of stream and static triples of the dataset. In this way, we used the dataset generator of LUMB [50], to realise different size of datasets, considering different universities: one (100572 triples), two (236336 triples), and five (643435 triples). From these three configurations, we extracted stream and static triples, to feed our stream reasoning systems. All values of the used dataset are summarised in Table 6.3.

Universities	Total triples	Stream triples	Static triples
1 uni	**100572**	**66774**	**33798**
2 uni	**236336**	**153179**	**83157**
5 uni	**643435**	**416923**	**226512**

Table 6.3 Dataset configurations

However, because these tests considered a fixed number of triples into the window, by increasing the number of stream triples, we increase the number of windows repeated into the stream, but not the number of simultaneously computed triples.

Figure 6.2 shows the results of our test where the throughput values of every single queries of the subset, executed in forward (identified by the radix *Forw_Q*) and backward (identified by the radix *Back_Q*). The forward and backward queries are reported in an ordered sequence, by alternating backward/forward, by starting from the backward query 1 (Back_Q1) to the last forward query 14 (Forw_Q14). Moreover, three principals trends have been identified, referring the query 1, query 5, and query 14, that are reported into the graph respectively with Q1, Q5, Q14.

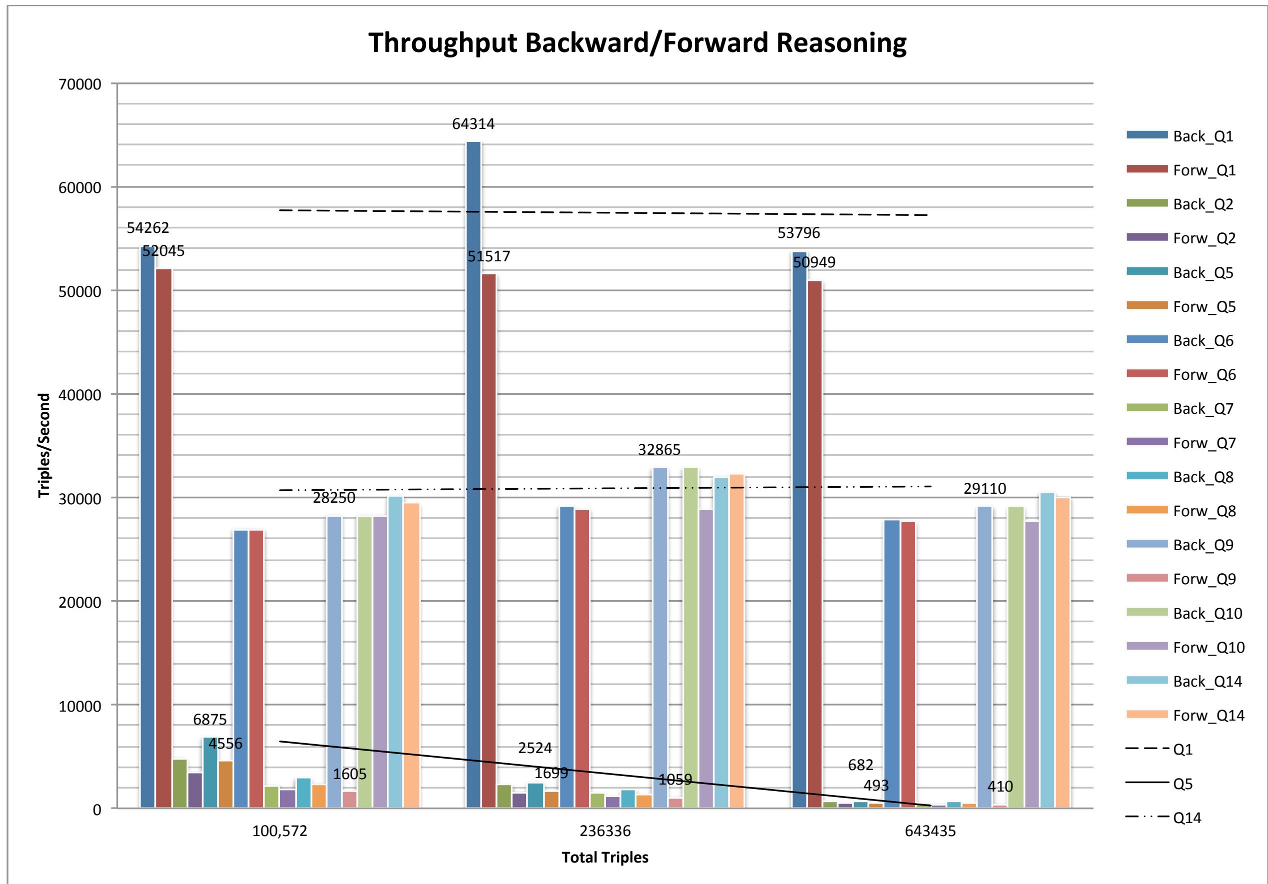


Figure 6.2 Forward/Backward reasoning approach

For sake of clarity of the results, we are made some remarks about the graph, by considering the three different classes of queries:

- the *stream/static queries*: they generally show the worst throughput values (with the exception of query 9 in backward reasoning: *Back_Q9*, that we will analyse later); that is explained well because they are the most complex queries of the set, since they unify stream and static reasoning;
- the *reasoning stream queries*: they present better performance of previous ones, because they need to consider only stream information, and they provide a less extensive and less complex set of triple patterns;
- the *no reasoning stream queries*: as was expected, they show the highest throughput values. That happened because the results set of the query if formed just by the tuples selected from the stream, without passing thought the stream reasoning modules.

It is also possible to see that, by increasing the number of input triples, the mains difference in terms of throughput values are verified just into the *stream/static*

queries (Q5 into the graph Figure 6.2), while all other ones do not providing any important changes. This happened because, even if the number of triples is enlarging, the number of triples that are compute simultaneously by the system is always the same and limited by the window size. However, for *stream/static queries* this does not happen. Indeed, they also use static triples materialised by static reasoner, that being static in nature, it does not exploit any window configuration. So, in this way, there is no limit, imposed by the window size, to the number of triples to calculate and all input triples are considered together at the same time.

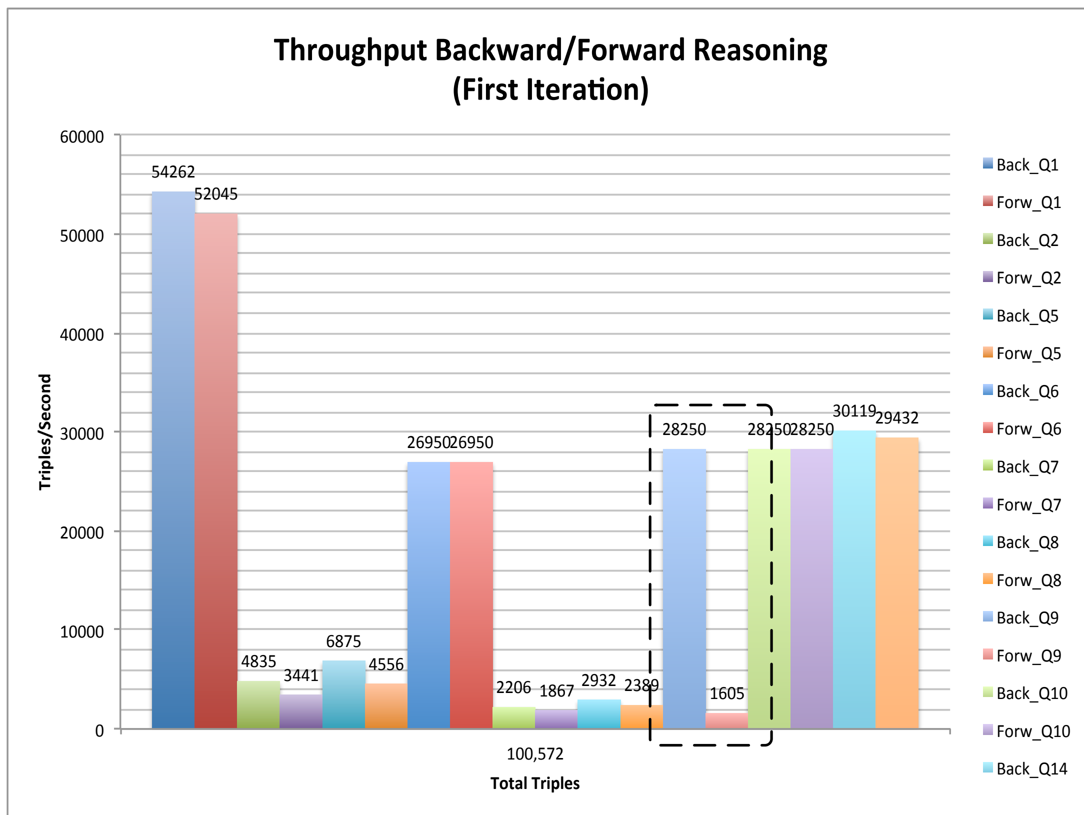


Figure 6.3 Forward/Backward reasoning approach (first iteration)

After having presented the general trend, we can focus on the difference between the forward and backward reasoning approaches, so we now consider only a single iteration from the previous graph (Figure 6.3).

As we were awaiting, the backward reasoning always performs the same or better than the forward approach; that happens because by design, the forward reasoning represents the worst case of the backward reasoning, since a backward reasoning can be considered to work in forward if there are no filters to be configured. It is also interesting to note that, even if the backward reasoning performs generally with

throughput around $1K_{triple/second}$ better than the relative forward reasoning approach, in the highlighted case of the graph of query 9, the improvement is around $26K_{triple/second}$. That happens because the combination of filters provided in the backward reasoning approach to specifically configure query 9 is able to discard a large numbers of triples that, otherwise, must be calculated by the same query working in forward reasoning.

6.1.2.2 Backward Stream Reasoning on different window sizes

In the previous subsection we have proved that the backward reasoning approach provide always better results than the forward approach. In this subsection we will continue testing only the backward stream reasoning approach, which we have used also in our use case of the Dublin city (Section 6.2). In this subsection we provide another interesting test by exploiting different tuple-based window configurations, since we need to evaluate the performance scalability of the system. However, being interesting in the performance evaluation of the backward stream reasoning, we avoid, in this case, the static component of the stream reasoning, and we focus on the performance provided only by the stream component. This time, we keep fixed the total number of triples and we vary the window size, in order to understand which relationship links the number of triples into the window and the throughput value of a query.

We exploit again LUMB, even if we do not consider the static part of the reasoning module between the throughput values provided in the graph; of course results are very different from the previous ones (Figure 6.2). Moreover, even if we exploit again the dataset with five universities (643435 total triples), we will consider only the stream triples (416923 triples) computed by the stream component. Thus, the final configuration for this second test of performance will consider 416923 fixed stream triples with five different window configurations respectively: 6677, 66774, 33387, 133548, and 66740 triples per window.

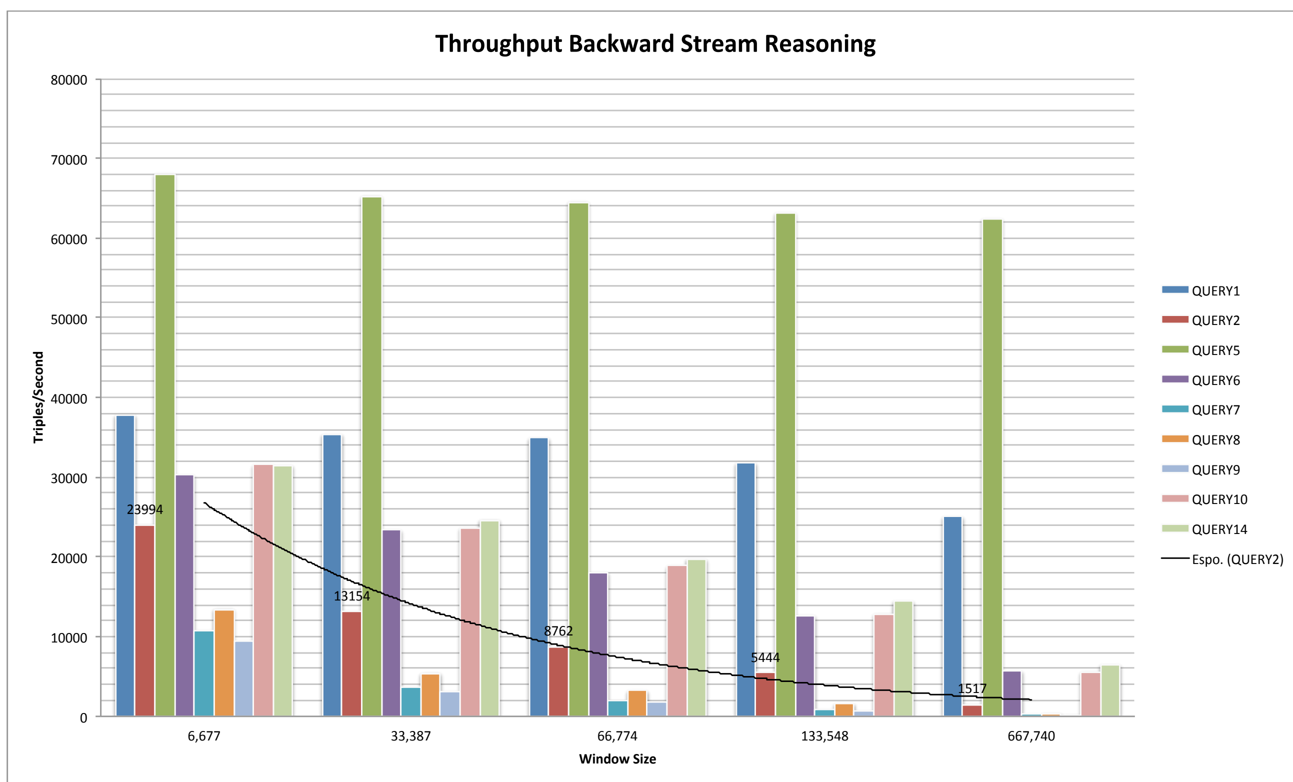


Figure 6.4 Backward stream reasoning performance on different window sizes

The graph shows that by reducing the size of the window the general throughput values of every query increase. On the other hand, by increasing the window sizes the general throughput values of the queries decrease. That the relationship between different throughput values and different window sizes is linear, as is the graph of Figure 6.4. To give a numerical case, we can consider query 2, which presents the same standard trend of all other queries. Thus, to the corresponding window size of 6677 triples, the initial throughput is $23994_{\text{triples/second}}$, and corresponding to the window size of 667740 triples the throughput decrease to $1517_{\text{triples/second}}$. This means that, increasing the window size of 100 times there is a reduction of the throughput value of 15 times, therefore the two concepts are relate in the rate 15/100. That is a very important experimental result, to size the window correctly for getting the expected throughput values.

6.2 DUBLIN CITY MONITORING

This section presents the use case where all defined components can be composed together, in order to provide a single execution framework able to evaluate heterogeneous data from a smart city.

With the proliferation of Closed-circuit Television (CCTV) systems for public safety and their decreasing cost, real-time monitoring of the video feeds, has become impossible to manually control all this data on the fly. In the Dublin city alone, there are approximately 200 CCTV cameras for public safety, and there is no possibility to manually monitor all these videos simultaneously. Therefore, our goal is to provide a method to select the best cameras to monitor in real-time the city "behaviour". In order to do that, we need to query real-time stream information from the city and perform this selection of cameras placed into the most interesting areas of the city. We identify the most interesting areas by considering the biggest changes that are happening into the city, starting from a regular situation.

In this scenario, we need to consider both data stream and static information, to have a complete description of the studied real-time environment, and also to give the correct attention to every single event that could happen into the city. In order to achieve this goal, we can specify a continuous query in DubExtensions and its relative translation into a complex SPL streaming application, executable on InfoSphere Streams.

We have tested the possible situation that we have described above by providing a complete simulation of its possible implementation. We have exploited a set of static and streaming data (private and public available), to describe Dublin city. Therefore, next subsection will provide a complete specification of the used data and the capabilities that the KPI query to realise the required computation (Section 6.2.1). Moreover, we also provide a complete simulation for the results of the KPI query in the Dublin smart city.

6.2.1 *Query and dataset definition*

As we have already said, we want to provide a query to compute a selection of the most interesting cameras in Dublin city. To do that, we identify the most interesting changes happening in a specific area, and consequentially select the most

appropriated cameras. However, this selection has to be computed with a specific DubExtensions query capable of:

- taking into account a number of stream measurements and to adding a score to each of them, weighted by the distance to the cameras. In this way, we can select the nearest cameras to the interested areas;
- assigning a score for the presence of sensitive facilities in the area (such as schools and hospitals), that can be useful in order to give a priority to the most important area of the city;
- detecting changes across two different window configurations: a short window of minutes to measure recent changes, and a second window of hours to consider the regular behaviour: that in order to take into account changes from a previous situation.

These are the capabilities that our query needs to provide in order to answer to the request: *which are the cameras nearest to the most interesting area of the city of Dublin?* In order to answer this question we have exploited a dataset, with static and stream information, obtained from public data available online and from IBM private information, to provide a vivid image of Dublin as a city of data (Section 6.2.2).

The stream information is compost of:

- *Dublin bus locations*: they are an incredible amount of data that provide the exactly location of 400 buses every 20 seconds, also expressing the current state of every bus (congested or flowing traffic). This information is provided by the *Service Interface for Real Time Information (SIRI)* standard protocol for real-time exchange information and it is not public available [49];
- *Pollutions level*: it describes the ambient air quality, picked up by 4 monitoring sites every hour. These data are expressed in CSV stream, they are public and provided by the Dublin City Council through *Dublinked* (http://www.dublinked.ie/);
- *Ambient noise level*: it describes the ambient noise quality, at a set of 7 monitoring sites every 5 minutes. These data are expressed in CSV stream, they are public and provided by the Dublin City Council through *Dublinked* (http://www.dublinked.ie/);

- *Dublin Bike usage*: this information describes the level of usage of 40 Dublin public bike stations of bike-sharing every second. These data are provided as CSV stream, they are public and available at (http://ocfnash.wordpress.com/2011 /02/02/dublin-bikes-revisi ted/);

These data contribute but are not enough to answer our query: we need to provide additional static information that could be used to give the fundamental positions of the cameras, the Dublin Bikes stations, and the amenities of Dublin city, which can be used to weight differently all possible areas of the city.

The static information is:

- *Camera locations*: they describe the exactly location of 192 CCTV cameras, for traffic monitoring, and public safety of Dublin city. These data are provided in RDF format, and they not public information;
- *Amenities locations*: they are 2626 hierarchically organised (described by an ontology) amenities of Dublin city, which have been extracted from *LinkedGeoData* (http://linkedgeodata.org/) by using a radius of 15km centred around the centre of Dublin (taking the geographical coordinates 53.3478, -6.25972 as a nominal reference point);
- *Dublin bike station locations*: they describe the location of 40 Dublin bike station, providing a description for every one of them. These data are provided as RDF, they are public and available at (http://ocfnash.wordpress.com/2011 /02/02/dublin-bikes-revisited/).

Both groups of static and stream information are summaries in the table below.

DATA STREAM	
Dublin Bus	600 buss positions every 20 seconds
Pollutions	4 monitoring stations, with level of pollutions provide every hour
Ambient Noise	7 monitoring stations, with level of noise provide every 5 seconds
Dublin Bike	40 Dublin Bike stations, with level of usage provide every second
STATIC RDF	
Camera locations	Locations of 192 CCTV cameras
Amenity positions	Locations of 2626 amenities
Dublin Bike station positions	Locations of 44 Dublin Bikes stations

Table 6.4 Complete dataset of the DubExtensions monitoring query

Moreover, because all amenities of Dublin are arranged in hierarchical structure, we exploit our backward static reasoner in order to realise the complete materialization of the amenities of Dublin city, organised as shown below.

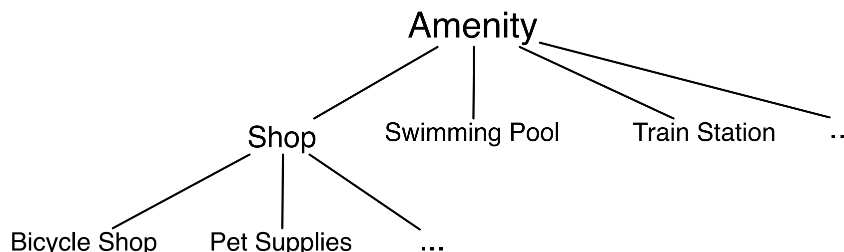


Figure 6.5 Section of the hierarchical organisation of the amenities of Dublin

The final query realised is really complex and too long to be entirely shown in this chapter, however we select some of the most interesting parts.

The query result clause

```
SELECT ?camera ( ( SUM(?bikechange)
                 + SUM(?busscore)
                 + SUM(?noisescore)
                 + ?amenitycount
                 + SUM(?pollutionscore) ) AS ?score )
       ?longCamera ?latCamera
       ( SUM(?bikechange) AS ?bike )
       ( SUM(?busscore) AS ?bus )
       ( SUM(?noisescore) AS ?noise )
       ( ?amenitycount AS ?amenity )
       ( SUM(?pollutionscore) AS ?pollution )
...
```

Figure 6.6 The KPI query results clause

The above figure shows the query results clause that provides the score of each camera, plus its latitude and longitude, and all intermediate results. The figure below shows a sample of the dataset specifications, with a selection of all different inputs and with all different window configurations (note that all windows have the same STEP, 20 min).

```
FROM CSV <Pollution_C0_2010_R.csv> 3 [RANGE 3h STEP 20m] AS 'pollutionC0short'
FROM CSV <Pollution_C0_2010_R.csv> 3 [RANGE 6h STEP 20m] AS 'pollutionC0medium'
FROM CSV <BallymunLibrarySoundMeter_R.csv> 3 [RANGE 1h STEP 20m] AS 'noiseBallymunshort'
FROM CSV <BallymunLibrarySoundMeter_R.csv> 3 [RANGE 2h STEP 20m] AS 'noiseBallymunmedium'
FROM CSV <Bus201001_R.csv> 0 [RANGE 40m STEP 20m] AS 'busshort'
FROM CSV <Bus201001_R.csv> 0 [RANGE 2h STEP 20m] AS 'busmedium'
FROM CSV <DublinBike_R.csv> 1 [RANGE 40m STEP 20m] AS 'bikestreamshort'
FROM CSV <DublinBike_R.csv> 1 [RANGE 2h STEP 20m] AS 'bikestreammedium'
...
FROM <Amenities_R.nt>
FROM <Cameras_R.nt>
FROM <DublinBike.nt>
```

The dataset defintion

Figure 6.7 The KPI query dataset definition

The Figure 6.8 presents some interesting parts of the extensive query pattern, where we use extensively used sub-queries to compute all intermediate results. In the listing below, one can see the weight of cameras locations based on the number of amenities near to the location of the cameras (the distance parameter: "0,01" has been calculate with experimental results).

```
{ SELECT ?camera ( ( COUNT(?a) ) AS ?amenitycount) ?longCamera ?latCamera
     WHERE {
          {?camera rdf:type <http://ibm.sctc.ie/dcc/camera/Camera> .
           ?camera  wsg84:long ?longCamera.
           ?camera  wsg84:lat ?latCamera.}
          {?a rdf:type <http://linkedgeodata.org/ontology/Amenity>.
           ?a wsg84:long ?along.
           ?a wsg84:lat ?alat.}
          FILTER  (((?along-?longCamera)*(?along-?longCamera) +
               (?alat-?latCamera)*(?alat-?latCamera)) < 0.01 ).
     } GROUP BY ?camera ?longCamera ?latCamera }
```

The query pattern - 1

Figure 6.8 the KPI query pattern - selection of the camera information weighted by the number of near amenities

In addition, Figure 6.9 shows another part of the query pattern that is another sub-query of the main query patterns that deal with the pollution CSV stream data. As one can see, it calculates the average of the normalised pollution level for every station, identified by "`pollutionid`" (the value 0 and 3.4 are the minimum and maximum level of pollution registered, in our data, and they are used to normalise the value)

```
{ SELECT ?pollutionid ( AVG( ( (?pollutionlevel - 0) / (3.4 - 0) ) ) AS ?
pollutionAvgS) ?pollutionlat ?pollutionlong
     WHERE {
          CSV 'pollutionC0short' {
               ?pollutionid ibm:csvCol_0 <Pollution_C0_2010_R.csv>.
               ?pollutionlevel ibm:csvCol_4 <Pollution_C0_2010_R.csv>.
               ?pollutionlong ibm:csvCol_2 <Pollution_C0_2010_R.csv>.
               ?pollutionlat ibm:csvCol_1 <Pollution_C0_2010_R.csv>.
          }
     } GROUP BY ?pollutionid ?pollutionlat ?pollutionlong } .
```

The query pattern - 2

Figure 6.9 the KPI query pattern - selection of the average of pollution level

The entire query has been further translated, by exploiting our system, and transformed in a SPL streaming application. Next figure provides a graphical

representation (provided by the IDE of InfoSphere Stream) of the SPL streaming application, in order to give an idea of the complexity and the number of SPL nodes that are necessary to translate our DubExtensions query (Figure 6.10).

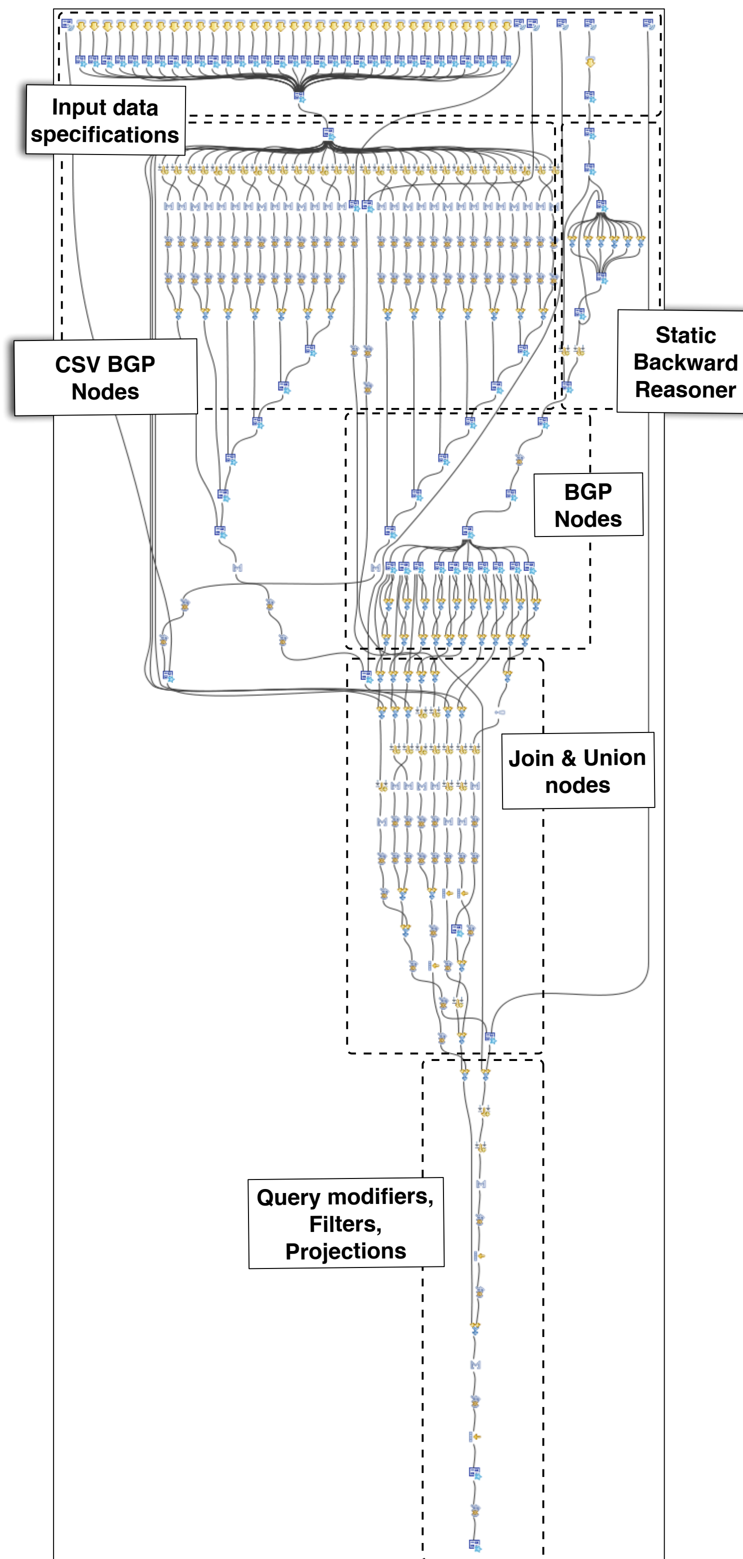


Figure 6.10 Graphical representation of the KPI DubExtensions query to monitor Dublin city

We have subdivided the KPI DubExtensions query in different parts that correspond to:

- *input data specifications*: contains all input CSV stream and static RDF, and it is responsible to feed the entire streaming application;
- *static backward reasoner*: responsible for the materialization of the hierarchical organization of amenities of Dublin city;
- *CSV BGP nodes*: responsible to manage all CSV streams and select, from the streams, necessary fields of the records;
- *BGP nodes*: responsible to manage the RDF static information, provided as output of the reasoner;
- *Join and Union nodes*: responsible to merge together CSV data stream and RDF information;
- *Query modifiers, Filters, Projections*: responsible to produce the necessary changes to the tuples results, in order to realise the final: aggregation, filtering, ordering, and projection.

6.2.2 Simulation

This subsection shows the results of the KPI query executed on the entire dataset previously defined. Moreover, to better give an idea of the complete real-time environment of Dublin city, we also provide a useful representation of all intermediate values exploited by the KPI query in selecting the CCTV cameras. In this simulation, by exploiting the previously defined conditions, the KPI query selects the ten most interested cameras every twenty minutes, so to highlighting the interesting events of the city in real-time.

Running the simulation, we see that in different period of time there are different interesting situation all around the city of Dublin, monitored by the CCTV cameras. However, during the time day, the most interested areas are all placed all around the city centre, that usually provide the most interesting conditions and from where the larger number of cameras are selected. That happens because it is the area that offers more amenities and more traffic congestions during the day. However, during the night, the points of interest are spread all over Dublin.

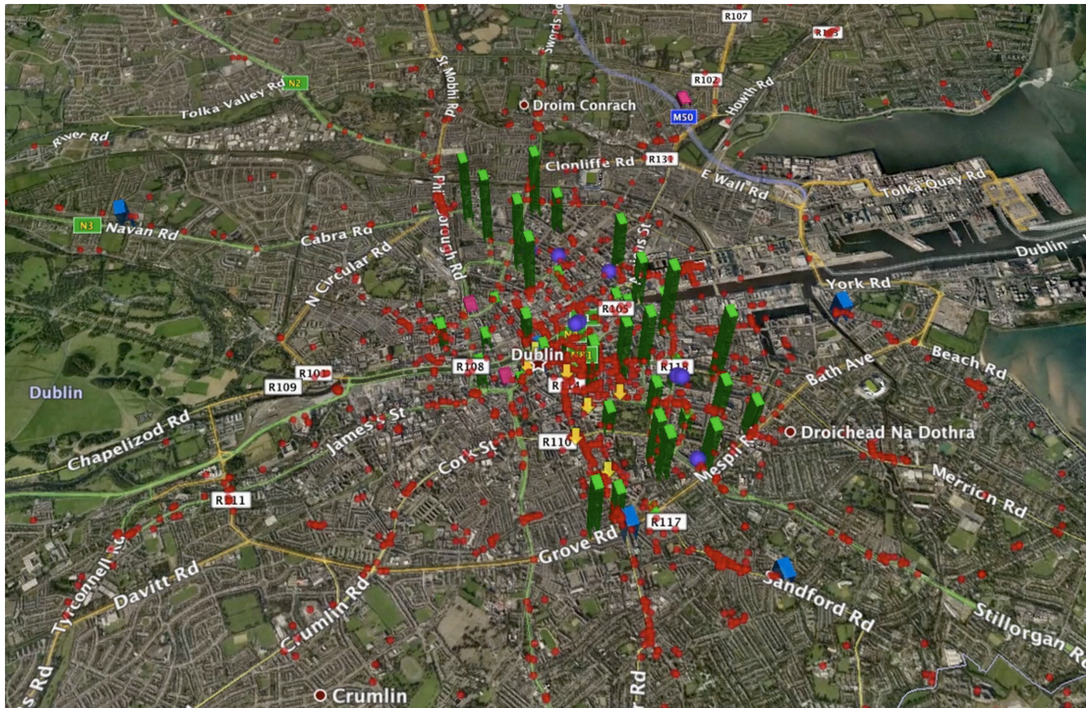


Figure 6.11 Screenshot of Dublin city simulation

In the above figure, we report a frame of the video realised from the simulation, with all different data provided in different colours and forms:

- the *smaller red points* represent the amenity locations;
- the *yellow rows* represent the camera locations;
- the *biggest purple point* represent the congested buses of Dublin;
- the *green polygons* represent the bikes locations with relative level of usage;
- the *blue polygons* represent the noise sensor locations with relative level of ambient;
- the *purple polygons* represent the pollution sensor location with relative level of air quality.

The entire simulation is long five days and it has been realised with Google earth (http://www.google.com/earth/index.html).

6.3 CONCLUSIONS AND FUTURE WORKS

The final simulation of the real use case represents an important achievement of our work, because we are able to easily integrate and query heterogeneous streams of data coming from a city, or more generally from a real-time environment. In this simulation, we have unified all components presented so far, and we have composed all of them together, in order to transform all these separated components in a unified tool, useful to provide an easy way to query real-time information. For this reason, the new SPARQL extension, the translation system, and the backward stream reasoning are the best achievements of our project. With our DubExtensions we have provided an easy way to define complex SPL streaming applications, able to run on InfoSphere Streams, a powerful business platform for data stream processing, and able to compute a complex sequence of elemental operators, in order to transform the tuples of a stream. DubExtensions has been defined from other two existing SPARQL extensions: C-SPARQL and CQELS, even if it has been further extended in order to work on heterogeneous data streams.

In order to pass from the DubExtensions query string to the related SPL streaming application, we have provided an automatically translator, that exploits a two phases translation, to simplify the query in SSE operators and to further transform these chain of operator in SPL nodes, that compose streaming applications. In this way, we have exploited a native approach as CQELS to obtain the complete control of the DubExtensions query execution, and providing a performant way to manage heterogeneous data streams. However, we did not focus our approach in increasing the performance of the stream system, we have not investigated the Joins order execution, because we have preferred to focus on those capabilities useful to operate with heterogeneous sources, as the definition of the new CSV BGP.

DubExtensions wants also to represent a connection between two different worlds: the Semantic Web vision and the Data Stream Processing, by providing the possibilities to query simultaneously RDF and CSV stream. For this reason, we have investigated backward stream reasoning techniques in order exploit the power of the backward reasoning approach over RDF stream. However, for the first implementation of our backward stream reasoning, we have exploited the set of thirteen RDFS rules by providing several simplifications, in order to obtain an easy and more limited subset of rules. By using this set of rules we have investigate the

dependencies relationships and filter configurations to reduce the number of triples needed for the materialization. However, by exploiting RDFS rules, we cannot deal with OWL ontologies. Even if this is a strong limitation for our reasoning system, we consider our approach as a first step through realise backward reasoning on streams.

The performance issues and the limited set of rules of the reasoner, represent our most important limitations, and the most interesting future extensions of our system. Since we did not investigate the Join order the execution plan produced by our translator could actually be the worst one, with a strongly impact with the general performance of the entire streaming application. Moreover, the set of rules that we have exploited is the simplest one and it cannot deal with realistic OWL ontologies, since the reasoning approach is not mature enough to materialise all information. However, we consider our reasoning approach a step forward in the investigation about backward stream reasoning. Since our work wants to give interesting points to future develop powerful backward stream reasoning. The whole system has been implemented with InfoSphere Streams that represents a fundamental part of the execution framework. It is a powerful business platform to process Data Stream, able to provide performant executions of our streaming application, but more important, the necessary configuration level to translate DubExtensions queries in a set of native operators.

Even if we have gone so far, this project need to be extended to overcome the limitations and to introduce optimizations, to be able to realise a complete and powerful execution framework, to provide real services to citizens of a smart city.

A first interesting extension of our work could be represented by the introduction of a more powerful set of rules that could deal with OWL ontologies, in order to overcome the limitation of our reasoning approach, by extending the number of RDFS rules.

A further improvement could be represented by a new matching algorithm able to find the better Join order as possible, by exploiting the execution history stand by the system.

Furthermore even if we are now able to execute on data coming from a city, our set of operators is still limited. Even if DubExtensions is able to express all possible queries that we could need, not all queries could be properly translated as SPL streaming applications, because their translation is not completely implemented. For

example at the moment we are not able to translate another query forms that is not a *SELECT* or graph pattern realised by the *GRAPH* clause.

We are aware that even if lot of work has been done, lot more has to be done, in order to reach the vision of a smarter city and integrate every kind of information in a more useful way. The integration of every possible heterogeneous sources of data, and their simultaneous consultation in real-time is the key to produce a smarter city to provide service on a human scale, able to simplify our everyday life in the city, and consequentially the quality of life of everyone.

CONCLUSION

In an integrated vision compulsory toward to realise a smart city, we need to find a way to manage the new complexity of the information that continuously flows in real-time from the city. Our goal was to create a new system able to merge and integrate two different worlds: Semantic Web and Data Stream, by exploiting heterogeneous real-time information, providing a way to query such data by exploiting Semantic Web techniques, and by managing heterogeneous data streams. In this way, our project represents a step forward the realisation of a smarter city that is able to provide a better quality of life to its citizens. In order to reach this goal, we have developed a system capable of processing heterogeneous stream information from the very large environment of a city, where the information is provided by heterogeneous sources in heterogeneous formats. Existing approaches are not able to directly manage such diversity, because they are still strongly related to specific data representation formats. For this reason, we have defined a new language able to deal directly with heterogeneous Data Stream and Semantic Web information, providing a way to query both types of data simultaneously and consequentially be able to answer with more useful and complete information.

Most available languages tend to follow a declarative approach used only to specify requests, but not able to realise the evaluation of its answers, which are performed by low-level applications natively able to manage Data Stream. For this reason, we have realised a translator, in order to transform the high-level specification in low-level application of Data Stream Processing.

Moreover, since our goal was to put together Semantic Web with Data Stream information, we have fatherly extended our system, to realise low-level applications capable of reasoning over the semantic meaning of Data Streams. In fact, we have optimised the standard approach reducing the computation level usually required, and consequently improving the general performance of the system.

These three items, high-level language, translator system, and reasoner, represent our best achievements to be exploited simultaneously in order to answer complex query evaluations and reasoning on real-time information coming from a smart city. Moreover, we prove our skills by describing the behaviours of the different areas of

the city of Dublin, by exploiting real data, provided in heterogeneous format. Our prototype has been developed in collaboration with IBM Research Dublin and it is part of a larger research project on Smarter Cities, focused on the research in water, energy, transportation, city fabric, risk, exascale computing, and marine environments [1]. As part of this large project our work represents an important step for querying real-time information from a city, providing more useful service to its citizens. However, our work is still subjected to further research in order to develop a more complete and optimised system.

Future extensions of our work are represented by a more powerful system able to process an incredible amount of real-time data from everywhere in the city, in order to be able to realise more complex analysis on an increasing amount of data, by improving every possible service of the city. Moreover, in a near future, these system will be available to everyone in order to simplify in an incredibly way our lives, by transforming every citizen in an active member of the community of its city. In this way, every citizen will be able to share useful information as reporting of problems, ideas, and necessities, that could help the entire community of a smarter city, in their everyday lives.

REFERENCES

1 IBM. IBM Research - Ireland. [Internet]. 2012 Available at: http://www.research.ibm.com/labs/ireland/index.shtml.

2 W3C Technical Architecture Group. Architecture of the World Wide Web, Volume One. [Internet]. 2004 Available at: http://www.w3.org/TR/2004/REC-webarch-20041215/.

3 W3C. Resource Description Framework (RDF) Model and Syntax Specification. [Internet]. 1999 Available at: http://www.w3.org/TR/1999/PR-rdf-syntax-19990105/.

4 W3C. W3C SEMANTIC WEB ACTIVITY. [Internet]. 2012 Available at: http://www.w3.org/2001/sw/.

5 W3C. LINKED DATA. [Internet]. 2012 Available at: http://www.w3.org/standards/semanticweb/data.

6 Oracle Corp. SQL, XQuery, and SPARQL. [Internet]. 2006 Available at: http://www.w3.org/2006/Talks/0301-melton-query-langs.pdf.

7 Richard C, Anja J. The Linking Open Data cloud diagram. [Internet]. 2011 Available at: http://richard.cyganiak.de/2007/10/lod/.

8 Gruber T Toward Principles for the Design of Ontologies Used for Knowledge Sharing. International Journal Human-Computer Studies. Academic Press. 1995;43(5-6).

9 Urbani J, van Harmelen F, Schlobach S, Bal H. QueryPIE: backward reasoning for OWL horst over very large knowledge bases. In: ISWC'11, editor. Proceedings of the 10th international conference on The semantic web - Volume Part I; 2011; Berlin.

10 Cherniack M, Balakrishnan H, Balazinska M, Carney D, Cetintemel U, Xing Y, Zdonik S. Scalable Distributed Stream Processing. In: CIDR, editor. CIDR 2003 - First Biennial Conference on Innovative Data Systems Research; 2003; Asilomar.

11 Barbieri D, Braga D, Ceri S, Della Valle E, Grossniklaus M. Stream Reasoning: Where We Got So Far. In: NeFoRS2010, editor. Proceedings of the NeFoRS2010 Workshop, co-located with ESWC2010; 2010; Heraklion.

12 Barbieri D, Della Valle E. A Proposal for Publishing Data Streams as Linked Data - A Position Paper. In: LDOW(2010), editor. Proceedings of the Linked Data on the Web (LDOW2010) Workshop; 2010; Raleigh.

13 Barbieri D, Braga D, Ceri S, Della Valle E, Grossniklaus M Querying RDF streams with C-SPARQL. ACM SIGMOD Record. 2010;39(1).

14 Chandramouli B, Ali M, Goldstein J, Sezgin B, Sethu Raman B Data Stream Management Systems for Computational Finance. IEEE Computer. 2010;43(12).

15 Krämer J. Continuous Queries over Data Streams - Semantics and Implementation. Dissertation. Marburg: Marburg, Philipps-Universität; 2007.

16 Diao Y, Franklin MJ. Publish/Subscribe over Streams. In: Liu L, Özsu MT.

Encyclopedia of Database Systems. Vol 1. Springer Publishing Company, Incorporated; 2009.

17 Arasu A, Babu S, Widom J The CQL continuous query language: semantic foundations and query execution. The VLDB Journal. Springer-Verlag New York, Inc. 2006;15(2).

18 Della Valle E, Ceri S, Barbieri D, Braga D, Campi A. A First step towards Stream Reasoning. In: Domingue J, Traverso P, editors. Future Internet Symposium - FIS; 2008; Vienna.

19 Battré D. Efficient Query Processing in DHT-based RDF Stores. Disertation. Berlin: Elektrotechnik und Informatik der Technischen Universität Berlin; 2008.

20 Kulkarni P. Distributed SPARQL query engine using MapReduce. Dissertation. Edinburgh: Master of Science Computer Science School of Informatics University of Edinburgh; 2010.

21 W3C. Rdf/xml syntax specification (revised). [Internet]. 2004 Available at: http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

22 W3C. RDF Test Cases. [Internet]. 2004 Available at: http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/.

23 W3C. Turtle - Terse RDF Triple Language. [Internet]. 2011 Available at: http://www.w3.org/TeamSubmission/2011/SUBM-turtle-20110328/.

24 W3C. RDF Vocabulary Description Language 1.0: RDF Schema. [Internet]. 2004 Available at: http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

25 Shaw P. RDF Schema. [Internet]. 2009 Available at: http://www.dur.ac.uk/p.h.shaw/teaching/ais/lectures/patricia/ais13-rdfs.pdf.

26 Guido N. Un sistema di tipi per SPARQL. Dissertation. Pisa: Informatica, Università di Pisa; 2011.

27 W3C. OWL Web Ontology Language - Overview. [Internet]. 2004 Available at: http://www.w3.org/TR/2004/REC-owl-features-20040210/.

28 Hogan A, Harth A, Polleres A Scalable Authoritative OWL Reasoning for the Web. International Journal on Semantic Web and Information Systems. Quarterly. 2009;5(2).

29 W3C. RDF Semantics. [Internet]. 2004 Available at: http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.

30 Urbani J, Kotoulas S, Maassen J, Harmelen VF, Bal H WebPIE: A Web-scale parallel inference engine using MapReduce. Web Semantics: Science, Services and Agents on the World Wide Web. Elsevier. 2012;10(0).

31 W3C. SPARQL 1.1 Query Language. [Internet]. 2012 Available at: http://www.w3.org/TR/2012/WD-sparql11-query-20120724/.

32 Oracle. Sql, xquery, and sparql. [Internet]. 2006 Available at: http://www.w3.org/2006/Talks/0301-melton-query-langs.pdf.

33 Prud EG. http://www.cambridgesemantics.com/semantic-university/sparql-vs-sql-intro. [Internet]. 2012 Available at: http://www.cambridgesemantics.com/semantic-university/sparql-vs-sql-intro.

34 Arias M, Fernández JD, Martínez-Prieto MA, de la Fuente P An Empirical Study of Real-World SPARQL Queries. CoRR. 2011;abs/1103.5043.

35 Barbieri D, Braga D, Ceri S, Grossniklaus M. An execution environment for C-SPARQL queries. In: ACM, editor. Proceedings of the 13th International

Conference on Extending Database Technology; 2010; New York.

36 Hebrail G Data stream management and mining. NATO Science for Peace and Security Series - D: Information and Communication Security. 2008;19.

37 Margara A, Cugola G. Processing flows of information: from data stream to complex event processing. In: ACM, editor. Proceedings of the 5th ACM international conference on Distributed event-based system; 2011; New York.

38 Jacques-Silva G, Kalbarczyk Z, Gedik B, Andrade H, Wu KL, Iyer RK. Modeling Stream Processing Applications for Dependability Evaluation. In: Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on; 2011; Hong Kong.

39 Arasu A, Babcock B, Babu S, Cieslewicz J, Datar M, K I, Motwani R, Srivastava U, Widom J. STREAM: The Stanford Data Stream Management System. Technical Report. Stanford InfoLab; 2004.

40 IBM Corp. IBM InfoSphere Streams Information Center. [Internet]. 2012 Available at: http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp.

41 IBM Corp. IBM InfoSphere Streams - Harnessing Data in Motion. [Internet]. 2010 Available at: http://www.redbooks.ibm.com/redbooks/pdfs/sg247865.pdf.

42 Hoeksema J, Kotoulas S. High-performance Distributed Stream Reasoning using S4. In: OrdRing2011, editor. First International Workshop on Ordering and Reasoning; 2011; Bonn.

43 Barbieri D, Braga D, Ceri S, Della Valle E, Grossniklaus M. C-sparql: Sparql for continuous querying. In: ACM, editor. Proceedings of the 18th international conference on World wide web; 2009; Madrid.

44 Le-Phuoc D, Dao-Tran M, Parreira JX, Hauswirth M. A native and adaptive approach for unified processing of linked streams and linked data. In: Springer-Verlag, editor. Proceedings of the 10th international conference on The semantic web; 2011; Bonn.

45 Le-phuoc D, Hausenblas M, Parreira JX, Hauswirth M, Dangan L. CONTINUOUS QUERY OPTIMIZATION AND EVALUATION OVER UNIFIED LINKED STREAM DATA AND LINKED OPEN DATA. Technical Report. Science Foundation Ireland; 2010.

46 IBM Corp. IBM Streams Processing Language Introductory Tutorial. [Internet]. 2011 Available at: http://publib.boulder.ibm.com/infocenter/streams/v2r0/topic/com.ibm.swg.im.infosphere.streams.product.doc/doc/IBMInfoSphereStreams-SPLIntroductoryTutorial.pdf.

47 Apache Software Foundation. ARQ - A SPARQL Processor for Jena. [Internet]. 2011 Available at: http://jena.apache.org/documentation/query/index.html.

48 IBM. Dublinked Datastore. [Internet]. 2012 Available at: http://www.dublinked.ie/datastore/datastore.php.

49 SIRI. Service Interface for Real Time Information CEN/TS 15531 (prCEN/TS-OO278181). [Internet]. 2011 Available at: http://www.kizoom.com/standards/siri/.

50 University L. SWAT Projects - the Lehigh University Benchmark (LUBM). [Internet]. 2012 Available at: http://swat.cse.lehigh.edu/projects/lubm/.