Politecnico di Milano

School of Industrial and Information Engineering

Department of Electronic, Informatics and Bioengineering

Engineering of Computer Systems

$\mathcal{H}$eaven: Supporting Systematic Comparative Research of RDF Stream Processing Engines

Advisor: Emanuele DELLA VALLE

Co-Advisor: Daniele DELL'AGLIO

Master thesis by: Riccardo TOMMASINI matr. 799120

Academic Year 2013-2014

To Aldo

and to my family,

thanks for all your support (and the fish)...

Acknowledgements

Milano, 1 Aprile 2015

Ringrazio il Professor Emanuele Della Valle, che ha reso la tesi un percorso incredibilmente formativo. È stato un lavoro impegnativo, ma sempre stimolante e divertente. Grazie per l'opportunità, il tempo dedicatomi e l'infinita quantità di suggerimenti ricevuti. Sono grato anche a Daniele Dell'Aglio, per la sua guida e l'aiuto tecnico e per aver reso stimolante ogni discussione. Un sentito ringraziamento anche a Marco Balduini, per il suo prezioso contributo, in un momento, per lui, sicuramente molto difficile.

Ringrazio la mia famiglia, per avermi sostenuto nelle piccole difficoltà quotidiane, che da solo non avrei mai potuto superare, e per avermi insegnato che vale sempre la pena impegnarsi al massimo.

Ringrazio gli amici, per non avermi permesso di abbandonarli del tutto e per aver condiviso momenti unici e di incredibile sincerità.

Ringrazio Teto. Cinque anni fa abbiamo iniziato insieme questa corsa e, come uno di famiglia, ci sei stato fino alla fine.

Ringrazio Francesco, per tutte le discussioni assurde che abbiamo fatto, i consigli onesti e per essere un vero amico da più di dieci anni.

Ringrazio Fabiana, per essere stata una guida spirituale, per avermi motivato e per aver sciolto nodi che stanno nella mia testa e nel cuore, non solo nei muscoli. Ringrazio anche Alberto, perchè lavora sempre dietro le quinte.

Ringrazio i Bomber, Fabio e Lorenzo. Per avermi mostrato come ci si diverte lavorando; per avermi dimostrato che chi punta in alto arriva in alto e soprattutto per avermi insegnato che "Nobody Works Like Us". Che dire ragazzi, se non "*we were stuck in a blender, and now we are saving lives. WHAT?*"

Riccardo

Abstract

Stream Reasoning (SR) research field is grown enough to prove that reasoning upon rapidly changing information is possible. RDF Stream Processing (RSP) Engines, systems capable to handle at semantic level RDF-encoded information flows, are increasing in number of implemented solutions. Now the Stream Reasoning community is working on the standardisation of the methods and tools that supported their development.

Many Computer Science (CS) research fields shown their interest for a deeper comprehension of their own work nature. Studies like [46, 51] investigated the publications in those field, highlighting that the majority of them are allied to an Engineering epistemology. However, they also evinced and criticised the concrete differences with other engineering research areas, which focus on evaluation of the proposed systems and not only on their design and development.

The lacks of an empirical approach can be ascribed to the complex nature of the software systems. However, it is possible to face such studies that can not be easily modelled, reducing the complexity of the analysis keeping intact the relevance of each involved system. In social science and economy, where researchers deal with cross case studies, it is commonly used a Systematic Comparative Research Approach (SCRA) within an experimental setting, which grants properties like repeatability, reproducibility and comparability to build the evaluation upon.

The SR community agreed that it is mandatory evaluating RSP Engines, understanding how these systems perform in real uses cases. Recent works in the filed [53, 41, 19] pursued this goal, providing benchmarks for RSP Engines evaluation. Further analysis pointed out the challenges involved by the Stream Reasoning research and posed the basis for a proper RSP Engines evaluation, describing in detail where previous works have failed and how the can be

improved [44].

The limitations of the existing benchmarking proposals proved that the empirical evaluation of RSP Engines is just at the beginning. What is still missing in an infrastructure that allows to compare, possibly automatically, the performances of many RSP Engines and that grants the properties of an experimental setting. In this thesis we brace this challenge borrowing from the aerospace engineering the idea of an engine test stand, which is an automatic facility for engine testing and development.

A test stand allows to design experiments and to execute them, evaluating engines in a controlled environment. Thus, we formulate the following research question: "*Can an engine test stand, together with queries, datasets and methods, support Systematic Comparative Research Approach for Stream Reasoning?*"

In this thesis we propose $\mathcal{H}$eaven, an open source framework that enables the Systematic Comparative Approach in the Stream Reasoning research field. $\mathcal{H}$eaven consists of: an RSP Engine Test Stand, which emulates the aerospace engineering facility in the Stream Reasoning context; the Analyser, which enables the Systematic Comparative Approach trough a set of methods and tools for the investigation, hierarchically organised into a stack; and, finally, four naive implementations of RSP Engines, called Baselines, which represent simple terms of comparison to start the comparative research upon.

Estratto

Lo Stream Reasoning è il settore di ricerca che ha dimostrato la possiblità di applicare procedure di reasoning su flussi informativi in rapido cambiamento. Un RDF Stream Processing (RSP) Engine è un sistema in grado di processare a livello semantico questi flussi, quando sono codificati secondo lo standard RDF. Il numero di RSP Engine implementati è in crescita e di conseguenza la comunità scientifica sta formalizzando i metodi e gli strumenti che hanno consentito lo sviluppo di queste soluzioni.

Diversi settori di ricerca nell'ambito della Computer Science, hanno mostrato interesse per una maggiore comprensione della natura del proprio lavoro. Sono stati fatti diversi studi che hanno analizzato i frutti della ricerca in questi settori [46, 51]. Questi hanno dimostrato anzitutto la natura ingegneristica di molte pubblicazioni nell'ambito della Computer Science, ma anche una discreta mancanza di valutazioni empiriche delle soluzioni implementate. Questa è una differenza evidente con le altre aree di ricerca legate al mondo dell'ingegneria, che si focalizzano su questo tipo di analisi.

Solitamente, nei settori informatici in cui la valutazione empirica è tralasciata, i sistemi proposti hanno una natura complessa e sfaccettata che è difficile da valutare. Tuttavia è possibile, con gli strumenti adatti, studiare anche casi complessi. Questo accade per le scienze sociali o l'economia, i cui soggetti d'indagine non sono di certo facilmente modellabili. In questi settori viene comunemente usato un approccio comparativo sistematico, che semplifica il problema di affrontare soggetti complessi, senza tralasciare gli aspetti che li rendono rilevanti. Questo approccio diventa applicabile solo in un contesto sperimentale appropriato, che garantisce proprietà come riproducibilità, ripetibilità e comparabilità.

La comunità dello Stream Reasoning ha colto la necessità di fornire strumenti per valutare correttamente gli RSP Engine, comprenderne il comporta-

mento e quantificarne il valore comparando le prestazioni in casi d'uso reali. Qualche passo in questa direzione è stato già fatto. Lavori recenti [53, 41, 19] hanno fornito framework di benchmarking per RSP Engine, mentre altri hanno posto le basi di queste valutazioni [44], mostrando quali erano le mancanze di tali framework.

Le soluzioni proposte si sono dimostrate limitate, e la valutazione empirica di RSP Engine è solo all'inizio. Quello che ancora manca è una infrastruttura che permetta la comparazione sistematica di RSP Engine, all'interno di un contesto sperimentale che goda delle proprità sopracitate. Per affrontare il problema, in questa tesi, abbiamo preso dall'ingegneria aerospaziale l'idea di un banco di prova, uno strumento di valutatione e sviluppo per motori.

Un banco di prova permette di progettare esperimenti ed eseguirli su qualsiasi motore, raccogliendo i dati per una successiva valutazione delle prestazioni. La nostra domanda di ricerca quindi è : "Un banco di lavoro per RSP Engine è la soluzione che permetta la ricerca comparativa e sistematica nell'ambito dello Stream Reasoning?"

In questa tesi proponiamo $\mathcal{H}$eaven, un framework open source per la ricerca comparativa e sistematica nell'ambito dello Stream Reasoning. Il framework si compone di un Banco di Lavoro, l'equivalente di quanto abbiamo visto nell'ingegneria aerospaziale ma per RSP Engine. Include quattro implementazioni naive di RSP Engine, dette Baselines. Questi sistemi semplificati permettono di iniziare la ricerca comparativa. Infine $\mathcal{H}$eaven contiene l'Analyser, un insieme di metodi di indagine e strumenti di supporto, organizzati gerarchicamente ed atti ad analizzare e comparare i dati raccolti attraverso l'esecuzione di esperimenti su RSP Engine tramite il banco di lavoro.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Stream Reasoning (SR) is a multidisciplinary research field. It focuses on developing and supporting methods and tools to continuously answer complex queries on a variety of fast flowing information. Example of queries across multiple social media stream are: "*What are the top five trend topics, under discussion, and who is driving the discussions in Dayton?*" or "*How a certain event in Milan influences the user activity?*". However, the application domains of SR are not limited to social media analytics only. Semantic interpretation of sensor data, traffic monitoring and stream data integration are all possible use cases for SR [50].

Stream Reasoning research aims of integrating data streams and reasoning systems to answer queries. It has already posed theoretical formalisations that go beyond DSMS, CEP [32, 33, 25] and it has defined good basis to semantically handle Data Stream encoded in RDF [49, 10]. Despite Stream Reasoning application domains are heterogeneous and wide, recent works have demonstrated that reasoning upon rapidly changing information is possible. Successful application of SR techniques were applied for sensor data stream integration [16, 34] and Social Media Analytics [7]

The number of implemented solutions is rising and the need of standards and evaluation method is consequently growing too. Nowadays the SR community[1] is focusing on the formalisation of data, protocols an methods for RSP Engine development, but also benchmarks and evaluation frameworks to compare the results are required.

[1] `http://www.w3.org/community/rsp/`

1.1 Related Works & Motivations

Stream Reasoning over RDF-Encoded information flows (formally, RDF Streams) has its foundations in DSMS and CEP research field and RDF reasoning theories and technologies. Nowadays, the community is focused on the formalization of three points: (i) a data model for RDF stream; (ii) syntax and semantics of an extensions of SPARQL for continuous query answering under different entailment regimes; (iii) a protocol to interact with an RDF Stream Processing Engine (shortly, RSP Engine).

RDF Stream standardization was developed in early works [50, 33] and recently extended [8]. Continuous extensions of SPARQL, like C-SPARQL, are mature and their development is proceeding [9]. Last but not least, many works in the field [53, 41] try to provide benchmarks and frameworks in order to evaluate all the RSP Engine implementations proposed.

What it is still missing is a systematic comparison of RSP Engines under repeatable conditions. As in many other fields of Computer Science, the researchers focused to model proposals and implementations only, lacking methods and tools to empirically evaluate complete systems [40]. A Comparative approach is required to improve SR research, which is being part of an engineering epistemology as many works have pointed out [46, 51].

Actually RDF streams, continuous queries, and performance measurements for benchmarking RSP Engines were proposed [41, 53, 19]. However the community still lacks an infrastructure for rigorous comparative research, which provides repeatability and reproducibility of typical of experiments.

1.2 Research Question

In Section 1.1 we present which lack affects the empirical evaluation of RSP Engines as a challenge for the current SR research. The number of the involved variables, together with the complex and multifaceted nature of RSP Engines, motivate the difficulties of conducting realistic evaluations, but it does not legitimate the lack. Research fields like economy [30], history [45], psychology and other social sciences [22], in which the subjects complexity is too high to be simplified into models and thus investigated, typically apply a Systematic Comparative Research Approach (SCRA).

Other engineering areas give a central role to empirical evaluation. The aerospace engineering, for example, enables experiments design, their systematic execution and the automatic comparison of results through the usage of *Engine Test Stand*. Such a tool allows an engine to be evaluated not only from an architectural viewpoint, but during the working state within a controlled experimental environment.

Existing queries, dataset and methods partially answer the problem of SR community to support SRCA on RSP Engines. It is possible to evaluate RSP Engines, but it is hard to make it systematically. Thus, we can formulate our research question as: "*Can an engine test stand, together with queries, datasets and methods, support SCRA for Stream Reasoning?*".

1.3 Heaven

This thesis work tries to answer the research question posed in Section 1.2: "*Can an engine test stand, together with queries, datasets and methods, support SCRA for Stream Reasoning?*". We propose the description of $\mathcal{H}$eaven – a proposal for an RSP Engine test stand, four baseline RSP Engines and the Analyser, whose aim is enabling rigorous comparative research of RSP engines.

$\mathcal{H}$eaven is a modular and extendible software environment for automated evaluating of RSP Engines.

The aerospace engineering inspired the development of a **Test Stand**, which allows design and run experiments over RSP Engines. The **Test Stand** accepts input RDF streams through the **Streamer** specific module; it gathers performance measures during the experiment execution and it saves this data through the **Result Collector** module, allowing post-experimental analysis.

The framework ensures the analysis through the **Analyser**, which consists of a set of methods and tools for the RSP Engine performances investigation. The Analyser methods describe how to drill down the analysis through different levels of details, while the tool-set allows to visualise, analyse and compare experiments w.r.t the required analysis level.

Last but not least, $\mathcal{H}$eaven also includes four **baseline** implementations of RSP Engines under the ρDF [39] entailment regime. The Baselines can be exploited as Simple, Eligible, Relevant and Elementary (Section 3.2) terms of comparison.

Finally, this thesis work brings an experimental evidence of $\mathcal{H}$eaven potential. We include an example of RSP Engine comparison evaluating $\mathcal{H}$eaven Baselines. The insights we gathered from those experiments demonstrate how $\mathcal{H}$eaven can lead to empirical evaluation of RSP Engines, enabling SCRA for the Stream Reasoning research field.

The entire $\mathcal{H}$eaven (i.e. the test stand, the four baselines and the analyser) are released open source[2] with the intention to foster comparative research of RSP Engines.

1.4 Outline of this Thesis

This thesis is organized as follow:

- **Chapter 2** contains an overview of the main research areas related to this thesis, like Semantic Web, Software Testing and Benchmarking. It also presents a background of the Stream Reasoning research field from the DSMS and CEP point view.
- **Chapter 3** describes the motivations that inspired our work. It formulate our research question and the requirements $\mathcal{H}$eaven must satisfy to successfully answer the question.
- **Chapter 4** introduces $\mathcal{H}$eaven design. It describes the *Test Stand* and the *Baselines* and the *Analyser* methodological approach.
- **Chapter 5** contains the details of $\mathcal{H}$eaven implementation. How we realised the Test Stand; how the baselines were developed and finally which tools compose the Analyser.
- **Chapter 6** describes the experiment design process we followed. It provides the evaluation of $\mathcal{H}$eaven Baselines as an empirical proof of the Test Stand potential.
- **Chapter 7** draws the conclusion of this thesis work and it proposes the future extensions of our research.

[2] `https://github.com/streamreasoning/HeavenTeststand`

Chapter 2

Background

In this chapter we include an overview of the main research areas related to this thesis. The Stream Reasoning research context, to which this thesis belongs, is introduced in Section 2.3. It has the aim of unifying the Semantic Web research area, we briefly present in Section 2.1 and the Information Flow Processing (IFP) research one, described in Section 2.2.

This thesis proposes $\mathcal{H}$eaven, an open source framework for the Systematic Comparative Research Approach. Section 2.4 contains the state of the are of the Empirical Research in the Computer Science. It is worth to note that Software Testing and Benchmarking are the two most relevant approach to empirically evaluate software systems. For this reason we include in this Chapter an introduction about Software Testing in Section 2.5 and one about Benchmarking in Section 2.6.

2.1 Semantic Web

The Semantic Web (SW) is the World Wide Web extension that provides a common framework to share and reuse data across applications, enterprise, and community boundaries. Tim Berners-Lee, the inventor of the World Wide Web and director of the World Wide Web Consortium (W3C), supervises the development of proposed Semantic Web standards. Nowadays, a large number of researchers and industrial partners collaborate with the W3C to the Semantic Web research.

"The Web can reach its full potential if it becomes a place where data can be processed by automated tools as well as people". [W3C, 2001].

The Tim's definition above highlights the Semantic Web vision of the new era, the Web 3.0, where the information can be processed and manipulated by machines. SW can give to the information a well defined meaning. The way to reach this goal is adding a new layer of meta-data to the existing documents and data. The semantic definition of data and documents enables to understand and respond to complex human needs. The outcome of this vision is an evolution to a new Web, where intelligent agents can browse data on behalf of the user and make knowledge easier to be reached.

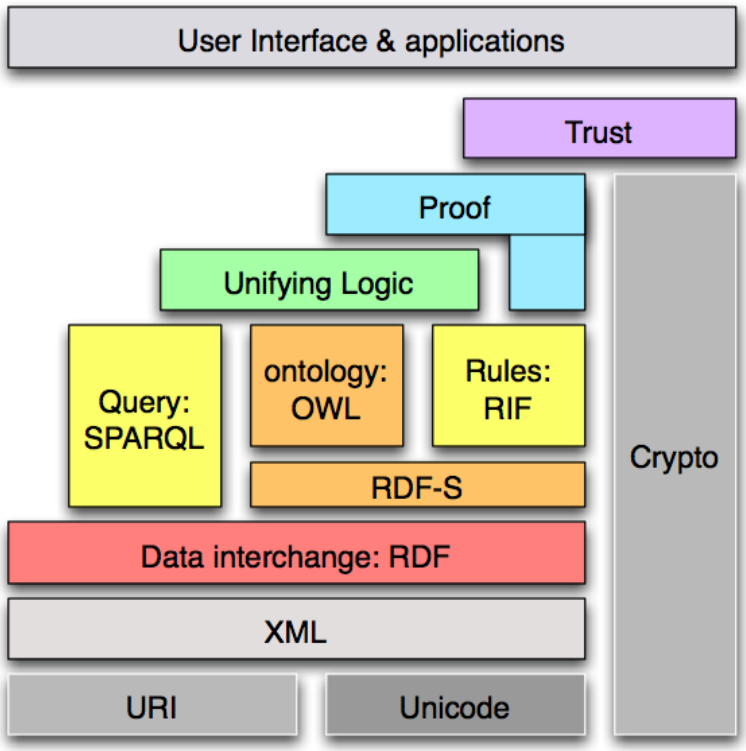


Figure 2.1 – Semantic Web Stack

Semantic Web world involves several technologies. Due to the complexity of the concept, the W3C proposed a layered approach, presented in Figure 2.1, in which each level contains a set of standards for semantic technologies.

The first level of the Semantic Web stack includes two standards of World Wide Web:

- The Uniform Resource Identifier (URI) and Internationalised Resource Identifier(IRI), which guarantee the interoperability between heterogeneous systems. They stay at the lowest level.
- The Unicode standard is a character encoding standard to represent and manipulate text in many languages.

On the top of them, the basic syntax level is provided by the eXtensible Markup Language (XML), a wide-spread standard for data exchange in the Web. The XML parsers support the lower-level syntax checking of the Semantic Web documents. However, the XML standard is not enough for the data exchange in the semantic web, because it is suited to recognize syntax, but not equivalent models. For this reason, three middle layers are required to ensure the semantic interoperability:

- The Resource Description Framework (RDF) is a language for expressing data models, which are usually encoded in XML.
- RDF Schema (RDFS) defines a vocabulary for describing classes and properties of RDF resources.
- The Web Ontology Language (OWL) extends the RDF Schema by adding advanced concepts in order to facilitate a more realistic representation of knowledge based on the RDF resources.

Last but not least, another important standard, which covers multiple layers, is the official Semantic Web Query Language: SPARQL, the official query language for the SW and a W3C Recommendation since January 2008.

In the following subsections, we will present some basic concepts of the Semantic Web: RDF in Subsection 2.1.1, OWL in Subsection 2.1.4, Linked Data in Subsection 2.1.5 and SPARQL language 2.1.6

Finally, technologies at the top layers are not yet standardized or implemented:

- The Rule Interchange Format (RIF) or SWRL allows knowledge inference from existing data.

- Cryptography is important to verify if semantic web statements are coming from trusted source.
- Trust involves the trustworthiness and reliability of the data.
- User interface is the final layer that will enable humans to use Semantic

2.1.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is the W3C standard for data interchange on the Semantic Web [31]. Through RDF we can describe a conceptual model of information in any given domain of knowledge. Information is represented in the statement form subject-predicate-object. In general a statement is composed by three different resources, indeed statements are also called triples and set of triples is called a graph. More formally, we provide the definition of RDF triples and RDF graphs:

Definition: Let I, B and L be three pairwise disjoint sets, defined as IRIs, Blank Nodes and Literals, respectively. A triple

$$(s, p, o) \in (I \cup B)I(I \cup B \cup L)$$

is an *RDF triple*, while a set of *RDF triples* is called an R*DF grap*h.

Notice that:

- IRIs are Internationalized Resource Identifiers, an extension of Uniform Resource Identifiers (URIs) that enlarge the character set from ASCII to Unicode. IRIs provide the linking structure of the Web and allow a globally connection between any different *RDF graph.*
- Literals are constant values, represented by character strings, which can be associated to a XML Schema datatype.
- Blank nodes represent anonymous resources, usually resources that group a number of other resources and which are never directly referenced.

The Resource Description Framework has several possible representations.

The most natural representation for triples is the N3 notation, which consists in explicit each triple in the form:

$$< subjectIRI >< predicateIRI >< objectIRI > /objectLiteral$$

The official syntax for RDF models is RDF/XML, which is an XML dialect for describing RDF. Many applications exploit the XML Language in the Web, indeed the XML document validation of the basic syntax of RDF/XML is possible through the RDF/XML standard schema.

```
1  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2                   xmlsn:rdfs="http://www.w3.org/2000/01/rdf-schema#"
3                   xmlns:foaf="http://XMLns.com/foaf/0.1/">
4    <rdf:Description rdf:ID="marvin">
5      <foaf:name>Marvin</foaf:name>
6       <rdf:type>
7          <rdfs:Class rdf:about="#ManicDepressiveRobot"/>
8       </rdf:type>
9    </rdf:Description>
10 </rdf:RDF>
```

Listing 2.1 – An example of a simple RDF/XML document:

The Code Snippet 2.1 contains an example of a simple RDF/XML document. Each RDF/XML document starts with an unique rdf:RDF element and the definition of the useful namespaces (lines 1-3). The rdf:Description tag (lines 4-9) represents an RDF Statement and it includes the subject of the statement, in our case Marvin. Inside the rdf:Description element we define two predicates: foaf:name (line 5) which contains an object literal, a string with the name of the resource; rdf:type (lines 6-8), a special predicate, which relates the individual resource to its class. In this case Marvin is an individual of the class ManicDepressiveRobot (line 7).

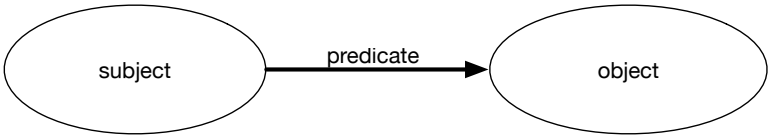


Figure 2.2 – RDF Model Graph

Last but not least, Figure 2.2 shows the graphical representation of an RDF model as RDF graph: subjects and objects are the graph nodes and proprieties are represented as edges connecting a subject to an object.

2.1.2 RDF Schema

RDF Schema (RDFS) is a semantic extension of the RDF. It allows to describe taxonomies of classes and properties, to define groups or restrictions of resources which are related and it enables to detail the relations between these resources. It is written in RDF and it also extends some RDF elements. Its constructs can be used to determine characteristics of other resources, such as the domains and ranges of properties.

RDFS class and property system is similar to the one of traditional Object-Oriented programming languages. RDF Schema describes properties in terms of the classes of resource to which they apply, through the domain and range properties. This property-centric approach aims of being easy-to-extend.

RDFS contains fifteen classes and the most relevant ones are:

- rdfs:Resource - everything in RDFS is an instance of this class.
- rdfs:Literal - sub class of rdfs:Resource represents a text string.
- rdf:Property - sub class of rdfs:Resource and represents the properties.
- rdfs:Class - sub class of rdfs:Resource, it represents the type concept typical of the OO programming languages and thus it is related to the rdf:type property. Every class is an instance of rdfs:Class.

It also contains sixteen properties, all instances of rdf:Property, and the most important ones are:

- rdf:type - used to state that a resource is an instance of a certain rdf:Class.
- rdfs:subClassOf - used to state that all the instances of one class are instances of another.
- rdfs:subPropertyOf - used to state that all resources related by one property are also related by another.
- rdfs:range - used to state that the values of a given property are an instance of one or more classes.
- rdfs:domain - used to state that any resource that has a given property is an instance of one or more classes.

2.1.3 ρDF

Efficient data processing in SW depends on the trade off between the expressiveness of the language that describes the data and the dimension of the dataset. Nowadays, developers have to face with scalability problems, due to the enormous size of data to process. For this reason, several RDF and RDFS fragments, with good trade off between performances and expressiveness, have been proposed.

ρDF [39], is a fragment of RDFS that conserves the original semantic and covers only those vocabulary and axiomatic information that serves to reason about the data it describes and not about the structure of the language itself. ρDF comprehends:

- rdfs:subPropertyOf.
- rdfs:subClassOf.
- rdfs:domain.
- rdfs:range.
- rdf:type.

Notice that w.r.t RDFS (see Section 2.1.2) it does not include the rdfs:Class, which is relevant for Semantic Web ontological level, but it does not play any relevant role in the frame of RDFS deductions. A similar argument can be done for rdfs:subClassOf and rdfs:subPropertyOf reflexivity, which do not play any relevant role in the semantics and thus can be avoided without having side-effects.

[39] provides the evidences that ρDF reserves the normative semantics and core functionalities of RDFS. Moreover, ρDF can be considered relevant from the Stream Reasoning research point of view, because several works in the field has already successfully chosen it as entailment regime [48, 35].

2.1.4 Web Ontology Language

The Web Ontology Language (OWL) is the W3C standard for describing rich and complex knowledge about resources, groups of resources, and relations

between resources, or more formally an *Ontology*. Tom Gruber define an ontology as *a specification of a conceptualization* [26]. Practically, it can be a set of Description Logic axioms which tries to represent a fragment of reality.

OWL is obtained extending the RDF Schema Language with additional language constructs and some constrain on the usage of RDF(S) ones. The aim of these constrains is to restrain the language to a subset of First Order Logic. Different constrains define different levels of expressiveness and complexity. The W3C has standardized three different OWL dialects called Profiles[1], in Dec. 2012. In the following we present them in order of increasing level of complexity:

- OWL 2 EL - this profile is suitable for ontologies with a huge number of classes to manage the terminology or which include complex structural descriptions. It works well for applications domains that have structurally complex objects. OWL 2 EL does not support negation, disjunction, and universal quantification on properties.

- OWL 2 QL - this profile is suitable for applications that aim to both represent database schemas and translate reasoning into queries. It can be used as an high level database schema language, because it is realized using standard relational database technology (e.g., SQL). It also provides many of the main features necessary to express conceptual models such as UML class diagrams and ER diagrams. OWL 2 QL does not allow existential quantification of roles to a class expression, property chain axioms and equality.

- The OWL 2 RL - this profile is tailored for applications that demand a good trade off between scalable reasoning and expressive power. Rule-based implementations of OWL 2 RL, under RDF-Based Semantics, can be used with arbitrary RDF graphs. An ideal way to enrich existing RDF data, especially when the data must be massaged by additional rules. OWL 2 RL disallows statements where the existence of an individual enforces the existence of another individual: for instance, the statement "every person has a parent" is not expressible in OWL RL.

[1] http://www.w3.org/TR/owl2-profiles/#OWL_2_EL

```
1  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
2    xmlsn:rdfs="http://www.w3.org/2000/01/rdf-schema#"
3    xmlns:foaf="http://XMLns.com/foaf/0.1/">
4     <owl:Class rdf:ID="ManicDepressiveRobot">
5       <rdfs:subClassOf rdf:resource="#Robot"/>
6       <rdfs:subClassOf>
7         <owl:Restriction>
8           <owl:onProperty rdf:resource="#hasMood"/>
9           <owl:minCardinality>2</owl:minCardinality>
10        </owl:Restriction>
11      </rdfs:subClassOf>
12    </owl:Class>
```

Listing 2.2 – An example of a simple OWL DL RDF/XML document:

OWL can be serialized in the RDF/XML syntax. In the Code Snippet 2.2, we use OWL to describe the Class of ManicDepressiveRobot, which is a Sub-Class of Robot that has at least two Moods. The owl:Class tag (lines 4-12) is a shortcut to represent an RDF Statement about a resource with a rdf:type of owl:Class. The subject of the statement is the resource ManicDepressiveRobot, which represents the class of individual Manic Depressive Robots. This statement has two predicates, both of the type rdfs:subClassOf. The first predicate (line 5) relates the OWL class ManicDepressiveRobot to its super-class, the resource Robot. The second predicate (lines 6-11) states that ManicDepressiveRobot is a subclass of the OWL restriction class, which contains all the individuals that are in relationship hasMood (line 8) with at least 2 resources (line 9).

2.1.5 Linked Data

The Linked Data concept indicates a set of best practices in publishing and connecting Semantic Web data. Bizer, the Linked Data creator, defines the term as follow:

> Linked Data refers to data published on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external readable, defined, data sets, and can in turn be linked to from external data sets [28].

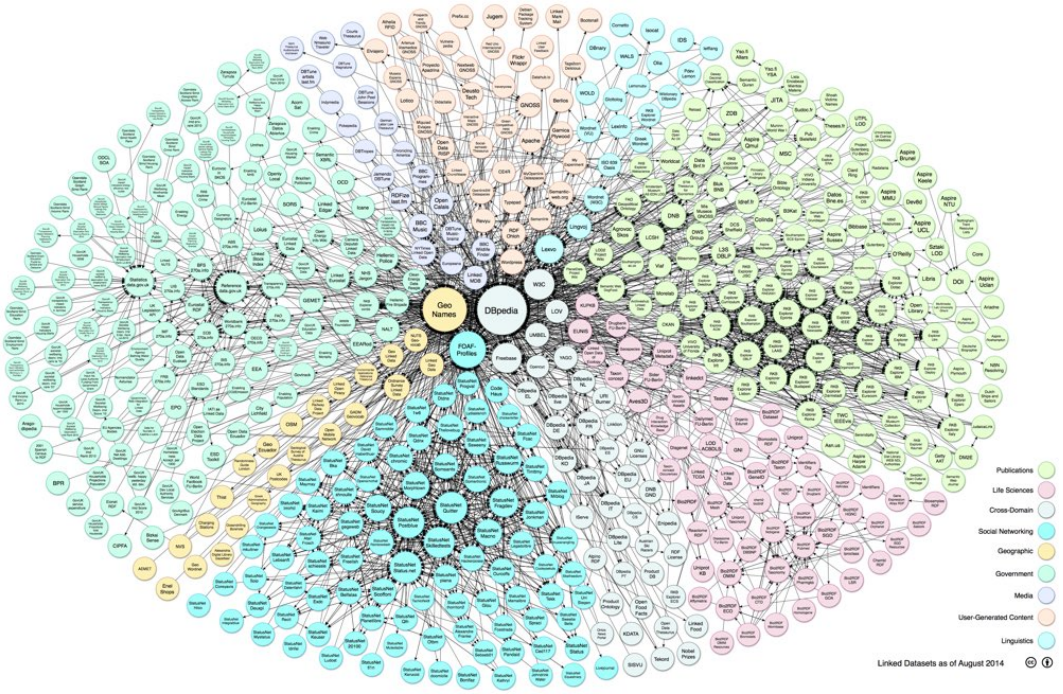


Figure 2.3 – Linked Data Cloud of August 2014[2]

Datasets that are published according to the Linked Data principles can be navigated using Semantic Web browsers. Figure 2.3 presents Linked Data Cloud updated to August 2014. It is the connected graph which comprises all the dataset published according to the LD principles. Each node of the graph represents a dataset, while each edge represents the connections between datasets.

The number datasets that follow the Linked Data paradigm is increasing every day and among the most important of them we can cite DBpedia, which wraps Wikipedia articles into a Semantic Web compliant form. Moreover, several governments, like the UK government[3], decide to publish their public data and connect them to the Linked Data cloud.

2.1.6 SPARQL

The SPARQL Protocol and RDF Query Language (SPARQL) are, since January 2008, the W3C standard query language for retrieving data in RDF format [42].

[3]http://data.gov.uk/

The SPARQL query language was initially designed to meet the requirements identified in the RDF Data Access Use Cases and Requirements. Only successively it has been formalized through the definition of an official algebra, developed as an evolution of previous RDF query languages such as rdfDB, RDQL, and SeRQL.

Recently, W3C released a working draft describing the new version of SPARQL[4]. SPARQL 1.1, this is the name of the new relase, is completely compatible with the SPARQL specification, but it has some new additional features, like aggregates, subqueries, negation and project expressions, i.e. the possibility to use expressions in the SELECT clause. These features were already part of several implementations of SPARQL, for example in the ARQ query engine.

SPARQL is defined as a graph-matching query language to face the RDF labelled graph. The language specifies four different query variations, each of which takes a WHERE Block to describe and restrict the searched graph:

- SELECT query: used to extract a set of variables and their matching values, called set of mappings in the table format.
- CONSTRUCT query: used to provide an RDF graph created directly from the results of the query.
- ASK query: used to provide a simple boolean value expressing if there are any matches in the graph.
- DESCRIBE query: used to extract an RDF graph based on the information related to the retrieved resources.

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT ?person ?name
3 WHERE {
4 ?person a foaf:Person.
5 ?person foaf:name ?name }
6 ORDER BY ?name
7 LIMIT 3
```

Listing 2.3 – An example of a simple SPARQL query

[4]http://www.w3.org/TR/sparql11-query/.

The code snippet 2.3 contains a simple example of SPARQL Select query. In general the structure of such queries (SELECT) consists of five clauses:

- PREFIX: The PREFIX keyword associates a prefix label with an IRI. A qualified name is a prefix label and a local part, separated by a colon, which is mapped to an IRI.
- SELECT: The SELECT keyword specifies the form of returning variables and their combinations by introducing new variable.
- FROM: The FROM keyword allows a query to specify an RDF dataset by reference.
- WHERE: The WHERE keyword provides the graph pattern to match against the data graph. The graph pattern can be in different form of simple graph pattern, group graph pattern, optional graph pattern, alternative graph pattern. Different keywords can be used in this clause such as OPTIONAL (for optional patterns), UNION (for unions of patterns), FILTER (for filtering patterns).
- Solution modifiers: this clause use different keywords like ORDER BY, LIMIT,. . . in order to modify the result of the query.

Further restrictions can be applied using keywords like FILTER, ORDER BY, LIMIT, AND, or UNION.

2.2 Information Flow Processing

The application domain of Information Flow Processing (IFP) includes systems able to collect information flows produced by multiple, distributed sources. Such systems, called IFP Engines, process the information in a timely way, extracting new knowledge from the information flow as soon as it is collected.

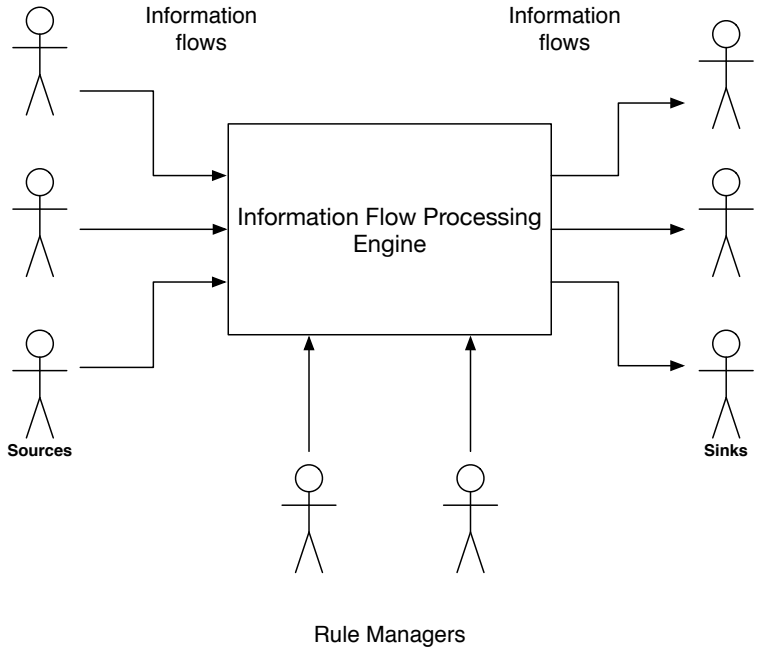


Figure 2.4 – The Information Flow Processing general Model proposed in [17] highlights the distributed and heterogeneous nature of information flows. The IFP Engine is a rule-based system that processes information flows. Practically it applies rules filtering, combining and aggregating different data flows, which have heterogeneous domain origins, in order to produce a new information flow that will be consumed by other systems.

An IFP Engine is *capable of timely processing large amount of information, which flows from the peripheral to the center of the system* as can be seen if Figure 2.4, which shows the general IFP Engine model proposed in [17]. The IFP Engine receives many input information flows and it processes the flow items, as soon as they are available, through a set of processing rules. The ruleset specifies how to filter, combine, and aggregate the different flows of information. Item by item the IFP generates a new flow, which represents the output of the engine.

The research works in the IFP context have developed two main models: the Data Stream Processing Model (DSMS) [4] and the Complex Event Processing (CEP) Model [36]. DSMS processing is led by two concepts, the Relational Data Stream and the Window. On the other hand, CEP processing exploits the notion of event and by considering the input flow as sequence of notifications, which can be aggregated to obtain catch the semantic value of the stream.

We detail these two models of IFP respectively for DSMS in Subsection 2.2.1 and for CEP in Subsection 2.2.2

2.2.1 Data Stream Management System

The DSMS research field views the IFP problem as an evolution of traditional data processing, describing it as *processing streams of data coming from different sources to produce new data streams as an output.* A Data Stream Management Systems (DSMS) extends the traditional DBMSs, in particular for the following considerations:

- streams are usually unbounded, while tables are not,
- no assumption on data arrival order and rate can be formulated and
- size and time constraints make hard, probable impossible, storing and processing data stream: one-time processing is the typical mechanism used to deal with streams.

DSMSs present also two main practical differences from traditional DBSMs: (i) DBMSs are designed to work on persistent data, while DSMSs are focused into transient data management, because data are continuously updated. (ii) DBMSs queries are run once and then return a complete answer. DSMSs queries instead are continuous queries, which continuously answer as new data arrives.

Figure 2.5 reports the general continuous queries architecture proposed in [6]. This model aims of highlighting several architectural choices and their consequences. A DSMS is modelled as a set of standing queries Q and one or more input streams, the DSMS can produce four outputs [17]:

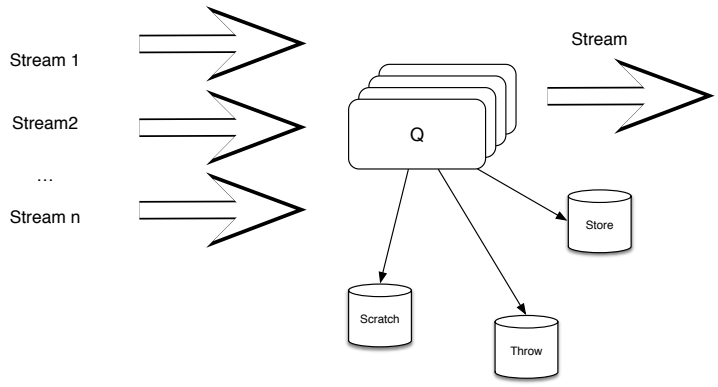


Figure 2.5 – The Continuous queries architecture proposed in [6] shows that a set of continuous queries may produce four different outputs, as a consequences of data streams processing: the *Stream*, which is composed by all the element of the answer that are produced once and never changed; the *Store*, which is composed by the a part of the answer that may change; the *Scratch*, which represents the working memory of the system; the *Throw*, a sort of garbage collector of stream tuples.

- The *Stream* - it is formed by all the elements of the answer that are produced once and never changed;
- The *Store* - it is filled with parts of the answer that may be changed or removed at a certain point in the future.
- The *Scratch* - it represents the working memory of the system. It is exploited to store data that is not part of the answer, but that may be useful to compute the answer;
- The *Throw* - it where the used unneeded tuples are collected and throw away.

Notice that the Stream and the Store together define the current answer to queries Q.

Moreover, the DB group of the Stanford University proposed the CQL stream processing model, which defines a generic DSMS through three classes of operators [2].

A DSMS system is is composed by operators that are able to manage the information stream in different phases. Figure 2.6 gives a representation of

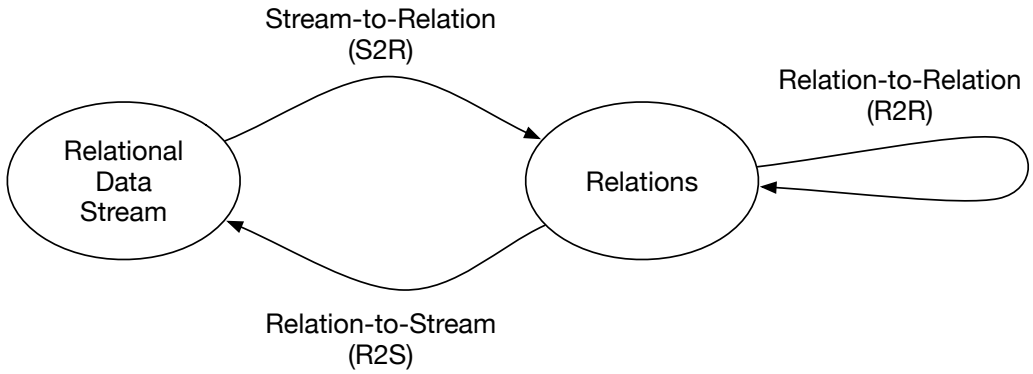


Figure 2.6 – The CLQ DSMS Model was proposed in [2]. The S2R operator transforms the incoming Relational Data Stream, which is possibly unbounded, into a bag of timestamped data items, we call Relations. The R2R operator applies relational algebra to the existing relations in order to transform into new ones. Finally, if the relations have to be transformed back to relational data stream, the R2S operator appends the results to the output stream.

the general model, composed by three modules that belong to three classes of operators:

The Stream-to-Relation (S2R) is the first class of operators that manage data streams. A stream is potentially unbounded, thus the S2R operators extracts finite bags of timestamped data items, transforming streams in relations. There are several operators of this class and the sliding window is one of the most studied. A sliding window creates a view over a portion of the stream that changes (slides) over time. Time-based sliding windows are sliding windows that create the window and slide it accordingly to time constraints: they are defined by two parameters, the width ω i.e. the time range that has to be considered by the window and the slide β i.e. how much time the window moves ahead when it slides.

The second class of operatos is the Relation-to-Relation (R2R). Relations may be transformed in another ones through relational algebraic expressions, which are applied by operators that belong to the R2R class.

Finally, the third class of operators is the Relation-to-Stream (R2S). This kind of operators are necessary when the output of the query processor should be a stream. Every time the continuous query is evaluated, the results are processed by the R2S operator, which determines the data items it has to

stream out and appends results to the output stream. There are usually three R2S operators:

Rstream, which streams out the computed timestamped results at each step. Rstream answers can be verbose as the same results can be computed at different evaluation times, and consequently streamed out. It is suitable when it is important to have the whole query answered at each step.

Istream streams out the difference between the timestamped results computed at the last step and the one computed at the previous step. Answers are usually short (they contain only the differences) and consequently this operator is used when data exchange is expensive. Istream is useful when the focus in on the results that are computed by the system.

Dstream does the opposite of Istream: it streams out the difference between the computed timestamped results at the previous step and at the last step. Dstream is normally considered less relevant than Rstream and Istream, but it can be useful.

2.2.2 Complex Event Processing

The Complex Event Processing (CEP) finds its origins in the publish-subscribe domain [21]. It comes from interpreting IFP items as notifications of events which arrive from external world. The traditional publish-subscribes systems process the incoming information flows one event at time, applying the filtering methods at topic level or at content level and evaluating the event relevance w.r.t the subscribers.

Complex Event Processing systems, namely CEP Engines, extend this behaviour, filtering and combining the events to understand what is happening in terms of higher-level information. The CEP model focuses on detecting occurrences of particular event patterns, which actually represent the higher-level information flows. For this reason, CEP Engines increase the expressive power of the subscription language to consider complex event patterns that involve the occurrence of multiple, related events [17].

Figure 2.7 shows the CEP Engine Model proposed in [17]. A CEP Engine is responsible for processing such events as explained above, in order understand what is happening in terms of higher-level information. Sinks aim to be notified about this new information level and thus they act as consumers of the CEP

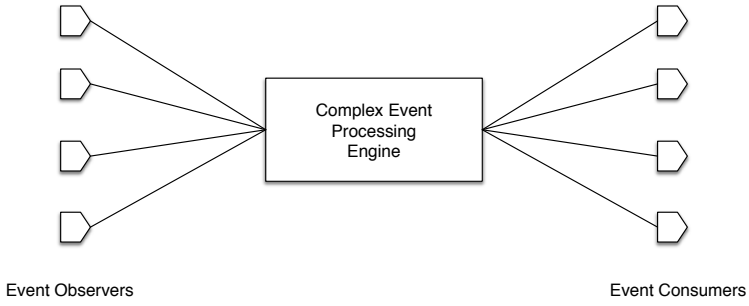


Figure 2.7 – The CEP model in figure is proposed in [17]. It relies on the ability to identify patterns that match multiple incoming events, which are seen as notifications. The processing consist in filtering and correlating the incoming events on the basis of their content. The goal is notify to subscribed sinks the presence of a certain pattern or some ordering relationships between events. The CEP Engine output is an event stream too.

Engine output events. CEP Engines are designed to face the main limitation of most DSMSs: the ability to detect complex patterns of incoming items, involving sequencing and ordering relationships.

2.3 Stream Reasoning

Answering such queries like *What are the top 10 emerging topics under discussion on Twitter, Facebook and Goolge+ and who is driving the discussions?* requires systems that can manage rapidly changing worlds at the semantic level. **Stream Reasoning** is a novel research trend focused on combining DSMSs, which can analyse data on the fly (see Section 2.2.1) and reasoners, which can perform such complex reasoning tasks (see Section 2.1), in order to make possible reasoning on rapidly changing information.

2.3.1 RDF Stream

Traditional DSMS system work with relational data stream, as the CQL general model in Figure 2.6 reports. In the Stream Reasoning context the data streams have to be semantically processed. Thus, the information flows require a semantic consolidation, which must consider both performances and

expressiveness. We focus *RDF streams*, which are data streams where the data items are modelled through RDF [5].

For example, let's consider the following stream: at time 1, an event e_1 states that an article *:paper*$_1$ is written by *:alice*, while at time 5, a data item e_2 states that *:paper*$_2$ is written by *:bob*. The stream $((e_1, 1), (e_2, 5))$ can be represented by the following RDF stream:

(:paper$_1$ *ub:publicationAuthor :alice), 1*

(:paper$_2$ *ub:publicationAuthor :bob), 5*

Initial works on RDF Stream formalisation have proposed two alternative formats for the Stream consolidation [49]:

- RDF molecules stream - it is an unbounded bag of pairs $< \rho, \tau >$, where ρ is a RDF molecule [20] and τ is the timestamp that denotes the logical arrival time of RDF molecule ρ on the stream;
- RDF statements stream, is a special case of RDF molecules stream in which ρ is an RDF statement instead of an RDF molecule .

The single RDF statement representation for events was exploited in different works [9, 33], while more recent researches [8] are studying the cases where events are modelled through RDF graphs (i.e. set of RDF statements).

The RDF stream definition is based on the relational data stream one. *RDF streams are sequences of timestamped data items, where a data item is a self- consumable informative unit.* Some the main characteristics of relational data streams still hold [5], thus RDF Stream are:

- **continuous**: new data items are continuously added to the stream;
- **potentially unbounded** : a data stream could be infinite;
- **transient**: it is not always possible to store data streams in secondary memory;
- **ordered**: data items are intrinsically characterised by recency.

[5] Cf. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

2.3.2 Continuous Extensions of SPARQL

There are many extension of SPARQL, which extends the SPARQL 1.1 (see Section 2.1.6) to include continuous operators (see Section 2.2.1):

Continuous SPARQL (C-SPARQL) is a language for continuous queries over streams of RDF data that extends SPARQL 1.1 [11].

For instance, the following C-SPARQL query asks to report every day the people mentioned during the last week in the stream of those who have published a paper:

SELECT ?person
FROM STREAM <http://www.ex.org/lubm-stream> [RANGE 1w STEP 1d]
WHERE {?person a ub:Person}

Query: 2.3.2: Example of C-SPARQL QUERY.

The requested information is not explicitly stated, but being the range of the *ub:publicationAuthor* property a *ub:Person*, an RDFS reasoner can deduce it and, thus, it can allow to answer the query.

Another example is CQELS-QL [33] a declarative query language built from SPARQL 1.1 grammar. As C-SPARQL, it extends SPARQL with operators to query streams. The main difference between C-SPARQL and CQELS-QL is in the R2S operator supported: CQELS-QL supports only Istream, whereas C-SPARQL supports only Rstream.

Last but not least, SPARQLstream [15] is another extension of SPARQL used in Morph_{stream}. Unlike CQELS and the C-SPARQL Engine, SPARQL-stream supports all the streaming operators presented in Section 2.3.4.

2.3.3 RDF Stream Processing Engine

Sections above report that RDF Stream Processing has three basic building blocks: 1) RDF streams, we introduced in Section 2.3.1 2) continuos extensions of SPARQL, like the ones we briefly described in Section 2.3.2, and 3) reasoning algorithms able to cope with rapidly changing information.

Collectively, IFP Engines that process RDF streams using those SPARQL extensions and reasoning algorithm are known as **RDF Stream Processing** (RSP) **Engines**.

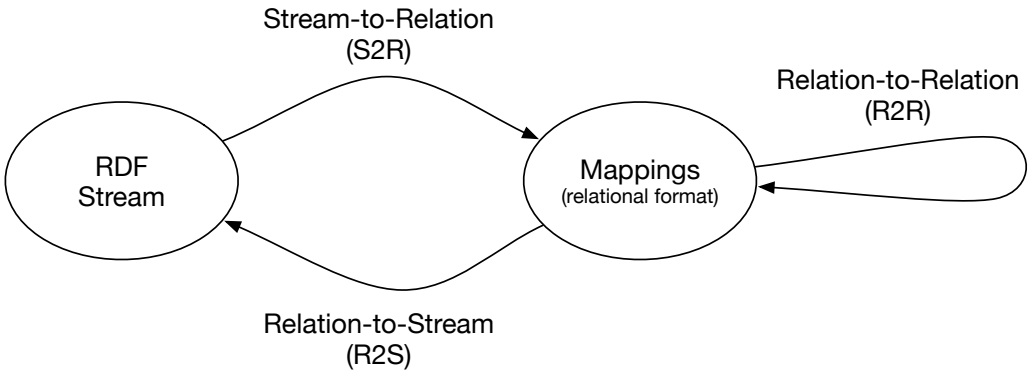


Figure 2.8 – This RSP Engine model is inspired to the CQL DSMS Model proposed in [2] and reported inf Figure 2.6. The incoming relation stream is now an RDF Stream. The S2R operator is usually implemented with a sliding window, which logically selects a portion of the RDF Stream. The evaluation of the continuous queries represents the R2R operator. The RSP Engine transforms the queries results back into an RDF Stream, applying the R2S operator.

Among RSP Engines, we target window-based RSP Engines, i.e. processors whose **continuous query language** allows to open sliding windows to capture segments of RDF streams. The C-SPARQL Engine[6], Sparkwave [32], CQELS [33], and Morph$_{stream}$ [16] are examples of window-based RSP Engines. The first two processes C-SPARQL [9], continuous queries, where as the second two proposes their own extensions (respectively, CQELS-QL and SPARQL$_{stream}$). The first two can also process C-SPARQL queries under RDFS entailment regime[7]. Morph$_{stream}$ uses Ontology-Based Data Access techniques for the continuous execution of SPARQLstream queries against virtual RDF streams that logically represent relational data streams.

The logical processing model of RSP Engines in Figure 2.8 follows the one initially proposed in DSMS (see Figure 2.6).

The following three logical steps[8] compose such model:

1. A portion of the RDF stream is logically selected using a sliding window operator; This is the **stream-to-relation** step.

[6] `https://github.com/streamreasoning/CSPARQL-engine`

[7] `http://www.w3.org/TR/sparql11-entailment/#RDFSEntRegime`

[8] Physical evaluation may differ from this logical steps.

2. The WHERE clause of the query is evaluated (including the possibly required reasoning tasks) and a sequences of results are produced, this is the **relation-to-relation** step.

3. Results are transformed back in timestamped elements and are appended to the output, this is the **relation-to-stream** step

Figure 2.9 proposes a block schema for RSP Engines, with the aim to clarify the general process presented in Figure 2.8. The schema disposes the S2R, the R2R and the R2S operators into a pipeline, drawing the RDF Stream path through the RSP Engine.

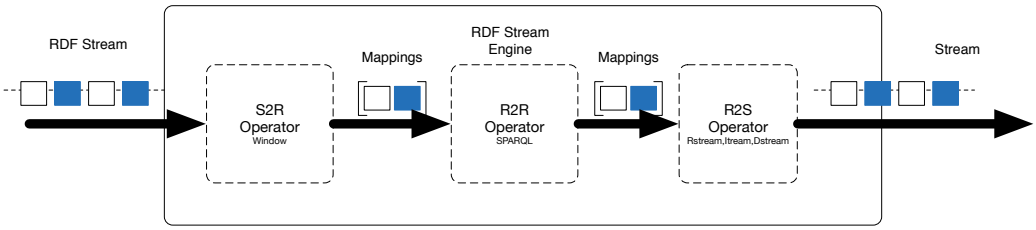


Figure 2.9 – This block schema translates the general RSP Engine model presented in Figure 2.8. It disposes each RSP Engine operator into a pipeline, clarifying the RDF Stream path: it is transformed into a set of relations upon which it is possible to perform continuous queries (S2R); the R2R operator applies a relation algebra which can handle data stream and produces new relations, then the query results are transformed back into an RDF Stream (R2S).

2.4 Empirical Research

The problem of understating the nature of the Computer Science (CS) research has one stronghold in Tichy's work of 1995 [46]. He and his collaborators evaluated 400 articles published in 1993, randomly selecting papers form ACM and from a few journals in Systems and Software Engineering. Tichy's research had the aim of classify Computer Scientist research habits. He proposed the following taxonomy [46] for the classification of the CS research works:

- *Formal theory*: theorems and their proofs, which provide formally tractable main contributions.
- *Design and modelling*: techniques and methods definition, models design and their implementation. Works like software tools or performance prediction models provide a contribution that cannot be proven formally.
- *Empirical work*: collection, analysis and observations about known techniques, systems, or models, or about abstract theories or subjects. The contribution of these works focuses on evaluation.
- *Hypothesis testing*: hypotheses definition and the design of experiments for their verification. The contribution of these papers is opening new research scenarios.
- *Others*: articles that do not fit any of the four categories above.

Tichy's evaluation showed that, despite the engineering epistemology of the majority of the evaluated paper, there was a deficiency in term of empirical evaluation w.r.t model *Design and Modelling* works. Researchers prefer to focus on the proposal of new models and their implementation rather than evaluating the existing ones. More recent works [51] observe and confirm Tichy's observations: the amount of evaluation reported in the Computer Science papers is still low.

Practically, an empirical study requires to define experiments for testing one or multiple systems. Comparing different results allows to distinguish what we believe it is truth from what we observe. The empirical evaluation has the aim of supporting the scientific method and rather plays a fundamental role in other research fields.

Software Engineering (SE) and many other Computer Science research areas have failed to produce the models and analytical tools that are common in other sciences [40]. CS research lacks the knowledge about the mechanisms which drive the costs and benefits of software tools and methods, making hard to understand whether analysis are based on faulty assumptions or the new evaluation methods are properly.

It is important to remember that empirical studies can be used not only retrospectively to validate systems or idea after they have been implemented, but also proactively to direct the research. Formal experiments, case studies and prototyping exercises are all adequate empirical studies. Independently from their form, they are a key way to fill the lack and sustains with credible observations those CS research fields which requires an evaluation of the proposals. Indeed empirical studies allow to learn something useful by comparing theory to reality through the following steps [40]:

- formulating an hypothesis or question to test,
- observing a situation,
- abstracting observations into data,
- analysing the data and
- drawing conclusions with respect to the tested hypothesis.

2.5 Software Testing

Software Testing (ST) techniques start with the intent of finding software bugs and ends with methods to evaluate system behaviour under testing conditions. In general ST is an investigation over software products to evaluate the software quality. According to IEEE Standards [29] there are two basic classes of software testing, black box testing and white box testing:

- Black box testing [BBT] - it ignores the internal mechanisms of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.
- White box testing [WBT] - is takes into account the internal mechanism of a system or component.

Black-box testing methods examine the functionality of a system without peering into its internal structures or workings. BBT exploits test cases built around specifications and requirements of the software. An external descriptions of the software is mandatory, and it must also include software specifications, requirements and design parameters. The test designer selects both valid and invalid inputs and determines the correct output without any knowledge of the test object's internal structure.

White-box testing method instead exploits complete knowledge about software internal structures. In WBT an internal perspective of the system is required to design test cases. The test designer analyses the code, understands the expected output and chooses many inputs to properly exercise the paths he has found.

The ST sub-field which tries to offer an objective and independent view of the software is Software Performance Testing (SPT), defined as *testing conducted to evaluate the compliance of a system or component with specified performance requirements* [29]. SPT requires to identify the key transactions and their data requirements. Once it is done, the test designer creates a number of different types of tests, in order to evaluate software performance w.r.t test variations. The design of the test follows the researcher needs about metrics evaluation. The choice depends on the nature of the application and how much time is available for performance testing.

The following testing terms are generally well known in the industry [38]:

- *Load testing* - it is the simplest form of performance testing, its aim is to understand the behaviour of the system under an expected load (the load kind depends on the software system). Moreover, it points to meet performance targets for availability, concurrency or throughput, and response time. This test will give out the response times of all the critical transactions and it can point out bottlenecks in the application software. Load testing is the closest approximation of real application use.

- *Stress testing* - it is used to understand the upper limits of capacity of a system and determine the system's robustness in terms of extreme load. A stress test may causes the application or some part of the supporting infrastructure to fail. The results of this kind of test are capacity measure as much as performance. It's important to understand software limitations, in order to face future growth of application traffic, which may be hard to predict.

- *Soak testing* - it is performed to determine if the system can sustain the continuous expected load. It essentially involves applying a significant load to a system for an significant period of time. The goal is to discover how the system behaves under sustained use or identify steady state conditions. During soak tests, memory utilization is monitored to detect potential leaks. Also important, but often overlooked is performance degradation, i.e. to ensure that the throughput and/or response times after some long period of sustained activity are as good as or better than at the beginning of the test.

2.6 Benchmarking

Benchmarking is the primary method for measuring the performance of a systems, hardware or an application. Indeed a benchmark is a procedure, problem, or test that can be used to compare systems or components to each other or to a standard [29]. Thus, benchmark results are used to evaluate the performance of a given system on a well-defined workload [37].

Many benchmark tests exist to evaluate a wide variety system or applications under different types of workloads. The user groups like the Transaction Processing Performance Council (TPC) [9], a non-profit corporation founded to define transaction processing and database benchmarks, or analogues corporations are useful resources of be informed about updated types of benchmarks.

Generic benchmarks allows the quantitative comparison of system performances or price/cost. In database context performance is typically a throughput metric (work/second) and price is typically a five-year cost-of-ownership metric [24]. The quantitative comparison requires the benchmark to be run on several different systems and to record each system is measurements. The estimation evaluated from results is usually the relative system performance, because the cost of implementing and measuring a specific application on many different systems is almost always prohibitive.

2.6.1 Domain Specifc Benchmarks

A single metric can not measure the performance relative to all applications of a computer systems [24]. Performances depend strictly on the application domain, because each system is designed for a few problem in a domain and may be inadequate to perform other tasks.

It is worth to note the work of Jim Gray about Domain-specific benchmarks [24], a kind of benchmarking methods and tools which responds to computer system diversity. A Domain-specific benchmark specifies a synthetic workload characterising typical applications in that problem domain.

In order to distinguish among several solutions and workload, Gray states four key criteria that a Domain-Specific Benchmark must meet to be useful.

[9] http://www.tpc.org/information

Thus, an useful Domain-specific benchmark must be:

- Relevant: It must measure the peak performance and price/performance of systems when performing typical operations within that problem domain.
- Portable: It should be easy to implement the benchmark on many different systems and architectures.
- Scalable: The benchmark should apply to small and large computer systems. It should be possible to scale the benchmark up to larger systems, and to parallel computer systems as computer performance and architecture evolve.
- Simple: The benchmark must be understandable, otherwise it will lack credibility.

2.6.2 Reasoning Benchmarks

The number of different reasoners available is increasing and many of them are already commercial solutions. Usually, reasoners are able to process very expressive ontology languages, which can represent rather complete knowledge. However, there is an high demand of less expressive ontology languages, which are less expensive in term of reasoning or other computational tasks. Commercial reasoners try to solve the issue called *computational cliff*. They face the trade-off between *complexity and expressiveness* versus *scalability*. Benchmarking tools are useful to evaluate reasoning system w.r.t. ontology languages [13], and the most important one is the Lehigh University Benchmark (LUBM) [27]. This work proposes a method for benchmarking Semantic Web knowledge base systems and provides an example of such a benchmarks.

Reasoning benchmarking environment requires:

- Ontology: LUBM exploits a synthetic ontology named Univ-Bench[10], which describes universities, departments and the activities that occur at them. The popularity of OWL is high many benchmarking tools allow testing performance with new ontologies [23].

[10]http://www.lehigh.edu/ zhp2/2004/0401/univ-bench.owl

- Workload: LUBM provides a random data generator, called UBA (Univ-Bench Artificial data generator), which creates extensional data over the Univ-Bench ontology [27].

- Queries: LUBM comes with fourteen testing queries to stress the reasoning capabilities over the workload. LUBM contains criteria for query definition which consider for example *Input size* or *Complexity* to stress the reasoner and evaluate performances.

Finally. LUBM contains three metrics for reasoners evaluation, which are commonly used in also database benchmarks:

- *Load Time* - the stand alone elapsed time for storing the specified dataset to the system;

- *Repository Size* - the resulting size of the repository after loading the specified benchmark data into the system;

- *Query Response Time* - it similar to databases benchmarking: for each test query the evaluation consist into 1) Opening the repository; 2) Executing the query for 10 times and computing the average response time. 3) Closing the repository.

LUBM contains two specific reasoning metrics: *Query Completeness and Soundness*: partial answering is possible in semantic web, indeed LUBM evaluate the *degree of completeness of each query answer as the percentage of the entailed answers that are returned by the system.* Moreover, LUBM also provides a metric for measuring the combination of query response time and answer completeness and soundness, to appreciate the potential trade-off.

2.6.3 DSMS & CEP Benchmarks

The Linear Road [3] is a stronghold about DSMS and CEP benchmarking. It posed several challenges about IFP system benchmark design which are:

- *Semantically Valid Input* - Input data can not be randomised but should have some semantic validity.

- *Continuous Query Performance Metrics* - Standard DBMS time to completion metrics is inadequate, it must also be considered *Query Response Time* and *Supported Query Load*.

- *Results Correctness* - benchmark implementations must be validated w.r.t the registered queries to ensure that results are consistent with the benchmark specifications.

- *Query Language Independence*: no standard query language for streaming systems exists, thus the query requirements must be language independent.

[3] defines the requirements a benchmarking system for IFP must fulfil according with these challenges. It provides also an implementation of a benchmark that meets all the requirements by design. The benchmark consists into a simulation of an urban highways system where toll charges are dynamically determined. The input data contains: a stream of position reports, which specify the location of a vehicle every 30 seconds; an historical query requests, which may be issued by a vehicle with some fixed probability every time it emits a position report.

Further works, like the BiCEP project[11], try to identify some o requirements for CEP benchmarking and they develop a synthetic benchmarking set to measure the event processing activity such systems. CEP Benchmarking are evaluated with Jim Gray criteria, we already presented in Section 2.6.1.

BiCEP project presents an first schema model for a CEP Engine benchmarking system. Figure 2.10 displays the schema: the input data are generated by the Event Generator Module; the CEP Engine results are consumed by the Answer Validation Module. Figure 2.10 shows also the CEP Engine interface, which ensures that any buffering, event cleaning or event transformation phase, that happens at the CEP Engine, is part of the overall performance measure.

Synthetic benchmarks offers many benefits like data availability, experimental control, and scalability. However, it is hard to develop a synthetic benchmark that is representative of such a wide range of CEP applications at the same time. BiCEP development is oriented towards a set of small domain specific benchmarks with different data sets and different queries [12].

[11] http://bicep.dei.uc.pt

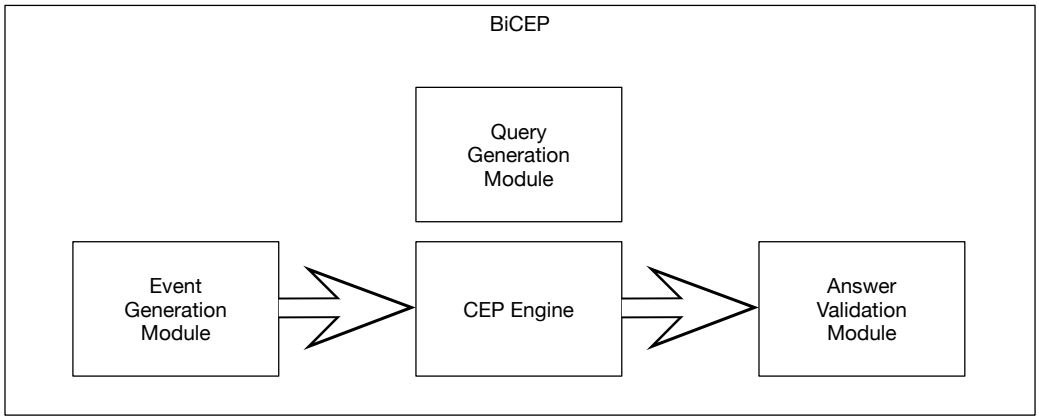


Figure 2.10 – BiCEP Benchmarking Schema proposed in [12]. *The Event Generator Module* produces the input consumed by the *CEP Engine* through the application of the queries provided by the *Query Generation Module*. The *Answer Validation Module* consumes *CEP Engine* results and states their validity. The CEP Engine interface ensures that any pre processing of the incoming events is part of the performance measurement.

Finally, BiCEP project presents a set of metrics for CEP benchmarking, the most relevant ones in performance evaluation context are: *Response time*: the time since the last event of some event pattern is fed into the system until the system notifies the event pattern detection. *Scalability* in order to compare system with different scale levels; *Adaptivity*. the system response to input variation, which is mandatory because CEP system rarely face stable input streams.

2.6.4 RDF Stream Processing (RSP) Benchmarks

RSP Engines are a clearly example of complex systems that face heterogeneous domains of application. This should be enough to follow the Tichy's suggestion and start their empirical evaluation. Moreover, the number of implemented RSP Engines is increasing, but the works about their evaluation are still limited and do not address properly the definition of the key performance indicators (see Section 2.3), making harder any comparison of the different systems.

Recent works on RSP benchmarking try to follow the Linear Road example, posing the main challenges for the SR benchmarking and developing them one by one into Seven Commandments for RSP Engine benchmarking [44].

In the following we report the these commandments as they are described in [44]:

RDF Stream Processing Engines must implement good strategies for *Load Balancing* [S.1], because they usually consider several input information streams with possible bursts. It is possible to stress the system under various conditions by repeatedly applying a set of changes to the input.

A stress test is required to measure the performance of *Flow Data Joins And Inference* [S.2]. It has to consider increasingly complex cascades of joins in order to design the stress test for Simple joins, which put no further constraints on the join but on the join-equality. The benchmark has to add further constraints on the joins, and subsequently to the data. In this way is possible: (i) enabling the testing og Sequential joins, which add a sequential constrains; (ii) allowing the testing of Temporal joins, which extend sequential joins by enabling advanced temporal constraints. On the other and, *joining stream and background data* [S.3] always results in simple joins. A stress-test must consider single joins and increasingly complex cascades.

The benchmark must test *Aggregates* [S.4], which enable counts, averages and any other arithmetic operation on groups of entities or literals. Moreover, it must take count of properly groups scaling: group number, group complexity or tuning the data to provide a big number of candidate, but a small number of selected groups.

In distributed settings, an RSP benchmark must ensure the correctness of query answers, handling *Unexpected Data* [S.5] like out-of-order arrival of information and data loss. A benchmark should be able to measure this ability evaluating precision and recall of the amount of missing data with two tests, (i) by increasing the number of out-of-order events or (ii) by testing how long and how many data can be handled until some nosily data observation will be no longer considered for processing.

A benchmark has two possibility for stressing the system through *Schema* [S.6] variations: (a) Evaluating the system ability to handle an increasing number of axioms in system ontology. (b) Changing statements that generate a more complex reasoning. It is important to know that: the axioms extension proposed by (a) could not have been deduced from existing ones and that the changes proposed by (b) may stress an RSP system significantly if they increase the expressive power of the schema, not only for the background data

but also of the data flow.

Finally, a benchmark should evaluate an RSP Engine through *Changes in Background-Data* [S.7]. Any stress test on background data changes should variate the update frequency and the entire amount of data involved in the update, forcing the system to access background-data from disk as much as possible.

Implemented proposals for RSP benchmarking has been published, but none of them impacts the community as the Linear Road did for DSMSs. More careful analysis of those solutions, presented in the following section, evidence some lacks w.r.t the commandments proposed in [44]

SRBench

SRBench is presented as *a general-purpose benchmark primarily designed for streaming RDF/SPARQL Engines and based on real-world data sets* [53]. The SRBench dataset is composed by: the LinkedSensorData, which is a real-world data set containing the US weather data published by Kno.e.sis[12]; GeoNames and DBpedia data sets, which allow to demonstrate the ability of the benchmarked system to deal with interlinked data.

Moreover, [53] provides some relevant metrics to evaluate the performance of the system:

- *Correctness of the query results* - it must be validated and the validation results should be expressed in terms of precision and recall.
- *Throughput* - it is defined as limit number of incoming data an RSP system is able to process per time unit.
- *Scalability* - it means evaluating how the system reacts to an increasing number of incoming streams and variation on number of registered continuous queries.
- *Response time* - it is the amount of time between a data item enters the system and the RSP Engine outputs the query results.

SRBench provides a query set composed by seventeen queries, which are designed upon a real use case in LSD. The queries vary involving single or mul-

[12]http://knoesis.wright.edu

tiple input streams, in order test many properties of RSP Engines. They cover the most important SPARQL operators and the common streaming SPARQL extensions and several queries require RDFS reasoning.

SRBench was criticised w.r.t. proposed commandments above [44]. It only cover [S.3] and [S.6] but with some limitations: the queries are fixed and thus do not allow an exhaustive assessment of join performance required by [S.3]. Testing variations of the expressive power is possible in SRBench, but it was not done yet as demanded by [S.6]. The commandments [S.2] and [S.4] are partially satisfied. The SRBench provides data and use-cases for sequential joins but it does not implement stress tests for temporal joins [S.2]. About [S.4], SRBench tests the aggregates only through single queries. The other commandments are not cover yet, even if some of them were marked as potential extensions of the current SRBench implementation. In [44], Table 1 summarises which features SRBench fulfil, which not, and which are only potential.

LSBench

LS Bench proposes methods that enables meaningful comparisons of RSP processing Engines and a framework that provides several customisable tools for simulating realistic data, running engines, and analysing the output. LSBench also includes three tests to evaluate the RSP stream Engines [19]:

- A functional test to verify the operators and the functionalities supported by the engines, similar to SR Bench.
- A correctness test is to verify if the tested RSP Engine produces the correct output. Actually the test assumes that the content of the output is correct, and it analyses only the number of produced answers.
- A maximum input throughput test, whose aim is evaluating the maximum throughput of the RSP Engine, by increasing the data streak rate and checking the number of the answers

[41] provides a data generator system for benchmarking, the S2Gen, which creates data set according to the *Social Network Data Stream Logic Schema*, presented again in Figure 1 of [41]. S2Gen allows to tune three parameters which influence the data generation task: 1) *Generating period*, the period in

which the social activities are generated; 2) Maximum number of posts/comments/photos for each user per week 3) the *Correlation probabilities*: there are various parameters for the data correlations which can be customized to specify how data is related according to each data attribute.

Finally, LSBench [41] provides for each test a set of 12 fixed queries. The queries in the set vary to address different features of the engines.

The Seven commandments for RSP benchmarking are not completely covered by LS Bench (see [44] Table 1). LSBench cover [S.3], but fixed queries do not allow to stress the system properly with the aim of evaluate join performances. [S.2] is only partially supported, because, LS Bench does not support stress testing for temporal joins but only for sequential. [S.4] is again only partially fulfilled: LS Bench tests aggregates only implementing single queries [44].

Benchmarking Query Result Correctness: CSRBench

Further works on correctness of the RSP Engine operational semantics, showed how this variates on different implementations and influence the query results correctness [19].

All Stream Reasoning benchmarks presented above have a common limitation: they do not check the correctness of output produced by the benchmarked RDF Engine. SRBench verifies only through functional tests the query language supported by the engines while, LSBench does not check the answers correctness, but it limits the analysis to the number of outputs. SR Bench and LS Bench make two assumptions on the tested systems which may be considered erroneous: (i) they work correctly; (ii) They all have the same operational semantics. However, these assumptions do not hold for RSP Engines in general and hence these benchmarks may supply misleading information about them.

An extension of the SR Bench for correctness checking, CSRBench, was proposed in [19]. It takes into account the issues that affect query results in their three main dimensions: system, query and input stream data. It includes in the benchmark three new types of parametrised queries, with the aim of stressing the S2R operators. Finally, it provides an oracle that generates and compares results of RDF stream processors and check their correctness.

Chapter 3

Problem Setting

In this chapter, we present the motivations behind our research work presenting the issues we aim to face. The chapter is organised in two sections: in Section 3.1 we introduce the comparative method, explaining why it is meaningful for the Stream Reasoning research, and we conclude presenting our research question. In Section 3.2, we formalise the requirements that any solution must satisfy to properly answer the research question we posed.

3.1 Comparative Research

Recent works, like [51, 46], have tried to classify the Computer Science (CS) research activity. They showed that the majority of the research works follows an engineering epistemology, in particular they belong to *empirical work* and *design & modelling* classes of Tichy's taxonomy (see Section 2.6.1). Despite what happens in other engineering areas, where the experimental research is almost dominant, for many CS research fields the proposals of new systems or models are more common then the evaluations of existing ones. Now the question is: *What are the motivation under this CS research lack?* The main problem in evaluating software systems or models regards their complex and multifaceted nature. Other explanations concern the difficulties of conducting realistic evaluations, because of the number of the involved variables.

A Systematic Comparative Research Approach (SCRA) is typically used in those fields where the complexity of the subjects goes beyond the possible observable models. This is the case of the social sciences [22] or history [45], which exploit techniques to deal with complex cases that can not be simplified

in experimental setting. The analysis of a single case study allows to deeply understand it, but it makes difficult to engage any form of generalisation. On the other hand, cross-case studies are more relevant and allow general thinking, but their final complexity represents a problem. We need a strategy that reduces the analysis complexity without losing the relevance of each involved system. In this regard, Russel Schutt discusses four stages to systematic qualitative comparative studies for history social phenomena:

S.1 Premise of the investigation: identification of possible causes.

S.2 Choose the cases to examine (location, language, gender).

S.3 Examine the similarities and the differences with shared methods.

S.4 Propose a causal explanation for the phenomena.

We will return on these stages later. Before, it is important to understand that this investigation method becomes meaningful inside the experimental environment. An *experiment* is a test under controlled conditions that is made to demonstrate a known truth or examine the validity of an hypothesis [1]. Complex cases are seen as a combination of known properties, which is possible to identify parallelism or state contrasts upon and which are used to set up experiment configuration. Researchers can exploit the notions of *reproducibility* to appreciate variations on changing experiment conditions, *repeatability* to consolidate observations through multiple identical executions and *comparability* to contrast the results to identify the differences.

Some Computer Science sub-fields attempted to lead case-driven analysis by the notion of experiment. Database community explores the idea of comparative research through benchmarking techniques and actually the quality of empirical studies is rising [51]. It is worth to note Jim Gray's work about transactional benchmarking (see Section 2.6.1) and Domain Specific Benchmarks (DSB). He states that *any comparison on performances starts with the definition of a benchmark or a workload*, but we need know the relevance of the metrics. From one application to another the performances frequently variate, because each system is thought to solve a small set of problems. A DSB must response properly to system diversities, by specifying a synthetic

[1] http://dictionary.reference.com/browse/experiment

workload to describe typical applications in the problem domain; moreover it must provide a performances workload on various systems and an estimation of relative performance on the problem domain. Gray proposes also four criteria that a DSB must meet, which are:

G.1 Relevance, it must measure the performance peak of systems when performing domain typical operations.

G.2 Portability, it must be easy-to-implement on many different systems.

G.3 Scalability, it must be meaningful for both small and large computer systems.

G.4 Simplicity, it must be understandable to obtain credibility.

Let's consider the relation between this criteria and Schutt's stages presented above. Gray states G.1 to identify the relevant metrics for the evaluation, as Schutt does in his first stage (S.1), which demands a pre-analysis phase of the phenomenon. Moreover, Social Science does not care about problem scaling, because those properties that define the case also determine the problem dimension (S.2). Gray poses the same concept in the DB context with G.2, demanding implementation-related conditions, but it also explicit the need to consider the dimension-related issues in G.3, because DB must consider the scaling problem. Last but not least, G.4 demands simplicity to obtain credibility while S.3 suggests to exploit those evaluation methods that are commonly accepted (shared) by the research community.

Motivated by the growing number of RDF Stream Processing techniques, the Stream Reasoning (SR) community strongly demands evaluation and comparison practices. RSP Engines, the systems that implement this techniques, have an high resulting complexity (see Section 2.3). Complex implementations of such system together with their execution semantic demonstrate that a meaningful comparison between RSP Engines is non-trivial. The interest on cross-case analysis between complex subjects draw the need of a comparative research approach. Initial efforts in this direction try to define frameworks that resume DSMSs [3] and Reasoning [27] benchmarking studies. LS Bench and SR Bench propose a set of queries, data sets and methods to test and compare existing RSP engines (see Section 2.6.4). Both these works share

a common background: the Linear Road Benchmark (LRB) is the only existing benchmark for relational data stream processing engines. LRB states the requirements DSMS benchmarking. However, LS Bench and SR Bench implement and extend this work without re-contextualising this requirements in SR context. Following discussions identify other lacks, i.e none of them completely face the problem of evaluating query result correctness, because they do not analyse engine semantics. LS Bench concentrates attention to the evaluation of engines throughput and it checks the correctness of the query result measuring the mismatch after comparing different RSP Engines. SR Bench entirely ignores this issue and analyses the coverage of SPARQL constructs for each commercial engine. More recent works describe deeply all the RSP Engine properties, identify the future challenges and provide a standardization benchmarking requirements: commandments for a meaningful testing on RSP Engines [44].

The SR community still lacks an infrastructure that can control the execution environment and allow to systematic testing. LRB provides a simulator to validate the benchmarked DSMS systems, but it does not face the problem of an online evaluation it. Researches in this area demonstrate that RSP Engine execution semantic is relevant [14], but does not evaluate its cost.

From the aerospace engineering we borrow the idea of an *Engine Test Stand*, a tool that allows experiments design, their systematic execution and automatic results comparison. An engine can not be evaluated only by an architectural viewpoint, it is necessary to understand its behaviour during the execution: *A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it*[2]. Thus, the community questioned *How to support SRCA on RSP Engines*? Now we have queries, dataset and methods, that partially answer such question and, thus, the new research one is:

> *Can an engine test stand, together with queries, datasets and methods, support SCRA for Stream Reasoning?*

The next section poses the requirements a proper answer to this research question must satisfy.

[2]The First Law of Mentat, quoted by Paul Atreides to Reverend Mother Gaius Helen Mohiam

3.2 Requirements

In order to enable the Systematic Comparative Research Approach on RSP engines through an Engine Test Stand, we need to answer the following questions:

Q.1 How can the behaviour of an RSP Engine be evaluated?

Q.2 What makes this evaluation rigorous?

Q.3 How can this rigorous evaluation be automated?

To answer Question Q.1, we exploit traditional definition of *experiment* presented above. We answer Q.2 referring to *reproducibility*, *repeatability*, and *comparability* of experiments. Through this concepts, it is easy to answer Q.3 formalising the requirements for the solution.

Reproducibility refers to measurement variations on a subject under changing conditions. We gather these conditions into experiment configuration, whose specification is up to the user. For this reason the solution must be independent from:

R.1 *Test data*, relevant RDF Streams and ontologies chosen from user's domain of interest.

R.2 *Query*, relevant queries registered from user's domain of interest.

R.3 *Engine*, any RSP Engine tested by the means of easy-to-implement software interfaces.

Repeatability refers to variations on repeated measurements on a subject under identical conditions. The solution must not affect the RSP engine evaluation, which, from a practical point of view, poses two requirements:

R.4 it must not be running when the RSP engine is under execution.

R.5 it must have reduced (and possibly constant) memory footprint.

Comparability refers to performance measurements nature and the relations between experimental conditions. The SCRA demands both the definition of *comparable metrics* and the standardization of *evaluation methods*, which means the solution must:

R.6 include *basic set of performance measurements* [44].

R.7 enable extensions of the basic set of performance measurements with user's new ones.

R.8 support performance measurements collection for further analysis.

R.9 allow *qualitative analysis* through tools for result visualization.

In terms of software engineering, any solution which satisfies the requirements above demands also some technical ones:

R.10 *Extendible Design*, the possibility to replace each module with one with the same interfaces, but different behaviour, without affecting architecture stability.

R.11 *Event-base architecture* to properly communicate with event-based systems like RSP Engines.

R.12 *Easy-to-Parse RDF Serialisation* for the events pushed to the RSP Engine in exam.

SCRA is case-oriented, it needs *initial terms of comparison* and *successful evaluations examples* to pose research guidelines. A baseline is an elementary solution for the RSP problem, which is relevant from an experimental viewpoint. Thus, to properly answer to the research question, we must have at least a baseline which satisfies the following requirements:

R.13 It must be Elementary: requiring the minimum design effort and being a naive solution without focusing on performances.

R.14 It must be Eligible: being a fair term of comparison w.r.t. available solutions.

R.15 It must be Relevant: covering one of the theoretical solutions.

R.16 It must be Simple: allowing to identify easily those characteristics which support hypothesis formulation and comparison.

Chapter 4

Heaven - Design

In this chapter we present $\mathcal{H}$eaven, an open source framework for Systematic Comparative Research Approach on RSP Engine. It consists in four Baselines and two main components: the Test Stand and the Analyser. Firstly, Section 4.1 introduces the Test Stand, which satisfies requirements from R.1 to R.8 and from R.10 to R.12, by executing experiments on an RSP Engine. Section 4.2 describes the Baselines, four RSP Engines that are included in $\mathcal{H}$eaven as naive terms of comparison, since they fulfil requirements R.13 and R.14. Finally, Section 4.3 presents the Analyser, which addresses requirements R.9 and R.10 allowing the user to visualise, investigate and compare experiment results.

4.1 Test Stand

Aerospace engineering defines an engine test stand as a facility used to develop, study and characterise engines. It allows to test operating regimes and offers measurement of several variables associated with engine process. A test stand may use actuators for attaining a specific engine state, which is a unique combination of the engine properties. The information collected through the sensors depends on the engine manufacturer, which usually provides his own stand or the facilities to test the engine with commercial solutions. The test stand executes black box testing over engines, because its users can not rely on the knowledge of engine internal mechanisms.

The definition above still holds its relevance in the SR context, with the difference that engines subject of the testing, RSP Engines, are IO-Systems.

An RSP Engine consumes RDF Streams and it produces new ones, by applying queries under some entailment regime and w.r.t. an ontology which does not change over time. Describing an RSP Engine means understanding the relation between input streams, the queries registered to it and what we call *operational semantics*, which requires to know the RSP Engine internal processes. Indeed, black box testing is the only possibility to analyse such system with a test stand. Even having access to the entire RSP Engine code, may result hard to characterise all the RSP Engine properties a priori.

In the following section we describe each elements that Figure 4.2 summarises. In Section 4.1.1 we describe the Test Stand pipeline and each module that composes it. In Section 4.1.2, we detail the Data Model exploited to represent experiments, events, query results and measurements. Finally, in Section 4.1.3, we describe the Test Stand workflow, representing in Figure 4.2 how it executes experiments to stress the RSP Engine that we want to test.

4.1.1 Modules

An aerospace test stand exploits different modules to simulate the operating regime for the engine in use i.e a module for fuel distribution, one for the engine mechanic support or to enable users interaction during the execution. Modularity allows to extend the test stand and specify testing procedure.

For the same reasons, requirement [R.10] demands to design $\mathcal{H}$eaven Test Stand as a modular system and, thus, it consists in the following three standalone modules:

- the Streamer, a source for the input RDF Stream;
- the RSP Engine we want to test;
- the Result Collector, a data acquisition system for both the query results and the gathered measurements.

The architecture of $\mathcal{H}$eaven Test Stand is represented in Figure 4.2. The Test Stand modules are arranged into a pipeline and communicate exchanging events [R.11].

The execution starts with the Streamer that hides the data generation logic in order to obtain data independence [R.1]. It pushes an RDF Stream

directly to the mounted RSP Engine. It is up to the STREAMER to respect [R.5] and not to influence the memory footprint with heavy data loading tasks.

An interface fulfils [R.2] and [R.3] (Query and Engine independence). It adapts the event flow to the RSP Engine in use and hides the query registration process, which happens at engine level and it is up to the RSP Engine provider.

The RESULT COLLECTOR is at the tail of the pipeline. It is part of the TEST STAND because the performance measurements are gathered during the execution together with the queries results data. The RESULT COLLECTOR is responsible to save this data at the end of each cycle without influencing the system. The evaluation usually happens a-posteriori through the Analyser (Section 4.3). However, real time analysis of the performance measurements are possible, but they may violate requirements like [R.4] and [R.5].

Last but not least, the TEST STAND has an external structure that sustains other modules and it provides APIs through which control the process. It gathers the data during the execution adding them to the query results. It controls the process ensuring that the TEST STAND does not run when the RSP Engine is running, as required by [R.4]. Finally, the *Test Stand Supporting Structure* allows the user (the RSP Engine developer) to develop a specific testing procedure for a given engine, extending the measurements set with new metrics, according to requirements [R.7] and [R.10].

4.1.2 Data Model

The Test Stand accepts as input an EXPERIMENT in the form of a tuple $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q}, >$ where:

- $\mathcal{E}$ is the RSP Engine subject of the evaluation [R.3];
- $\mathcal{D}$ is the input dataset [R.1];
- $\mathcal{T}$ is the ontology [R.1];
- $\mathcal{Q}$ is the query to be continuously answered by $\mathcal{E}$ [R.2].

From an experimental point of view, which metrics we sample during the execution have a different influence on the measurements. For example, asking to the system for the memory usage may influence the latency calculus or saving on disk the query results may influence the memory footprint. Thus,

we define there main kinds of experiment, which can also be combined, distinguishing on the data we want to sample and save. The choice of the experiment kind depends on the goal and the error tolerance of the research.

- Query Experiment, query results are saved on file.
- Latency Experiment, only latency metrics are saved on file.
- Memory Experiment, only memory metrics are saved on file.

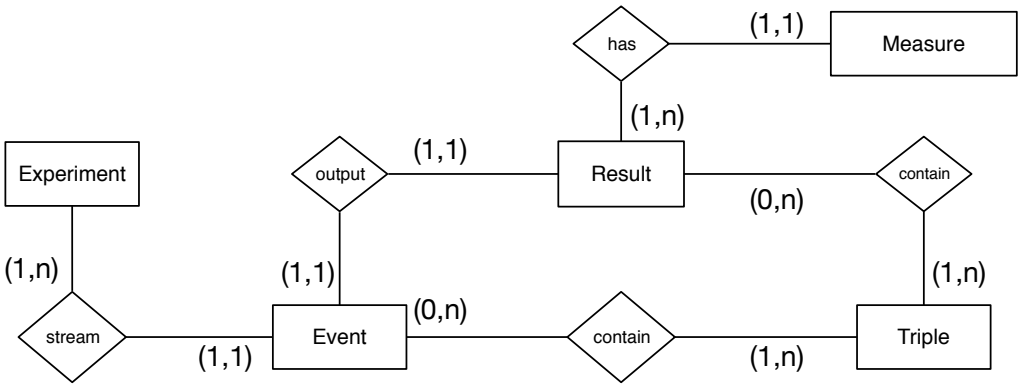


Figure 4.1 – Test Stand Data Stream ER-Diagram. The entities Event and Result represent unique events within an Experiment. Their relationships with Triple is described in two many-to-many relations. Result is also related to Measure by another many-to-many relation.

In order to describe the Test Stand Data Model, Figure 4.1 shows its Entity-Relation diagram. The entity attributes are not reported to simplify the interpretation, but they are included in the relative Logic Schema.
Experiment(ID, Timestamp Start, Timestamp End, Engine, Ontology, Query, Dataset, Description)
Event(ID, Experiment ID, Timestamp)
Result(Result ID, Experiment ID, Event ID)
Measure(ID, Value)
Measurement Set(Measure ID, Result ID, Experiment ID)
Triple(S,P,O)
Output Triple(Result ID, Experiment ID, S, P, O)
Input Triple(Event ID, Experiment ID, S, P, O)

The Experiment entity contains the metadata of the tuple $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q}, >$, which semantic is explained above. "Timestamp Start" and "Timestamp End" are relevant metrics for further analysis and system control.

An Event is unique inside an Experiment, it is possible to send two events with the same timestamp and identical tripleset. The Timestamp field allows to order events after the execution.

A Result is associated with one and only one Event. It contains the results to the engine queries w.r.t the active window and the set of the measure gathered during the execution.

The Measurement Set table represents the many-to-many relation between the Result and a number of measure that may variate to fulfil requirement [R.7] (extendible measurement set).

We include the concept of the Triple in order to model the content of Event and Result. Input Triple and Output Triple are the tables which represent two many-to-many relations, respectively between Triple and Event and Triple and Result.

4.1.3 Workflow

The Test Stand orchestrates the communication between the upstanding models, forcing the Streamer to push events to the RSP Engine and the Result Collector to listen the output and collect the results. To explain the Test Stand workflow, we split the process at the points when the modules exchange events. Indeed, each message represents a different logic step in the experiment execution cycle.

Six different steps are identified by six events exchanged by Test Stand, Streamer and RSP Engine. The Result Collector only receives events, terminating each cycle.

In step (1), the Test Stand takes the experiment and starts the execution. It executes the experiment $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q} >$ stressing $\mathcal{E}$ for a certain period of time looping through the steps from (2) to (5) illustrated in Figure 4.2.

In step (2), the Streamer pushes to $\mathcal{E}$ an event CTEvent. This event is a portion of an RDF Stream picked from the data $\mathcal{D}$ and it consists of a set

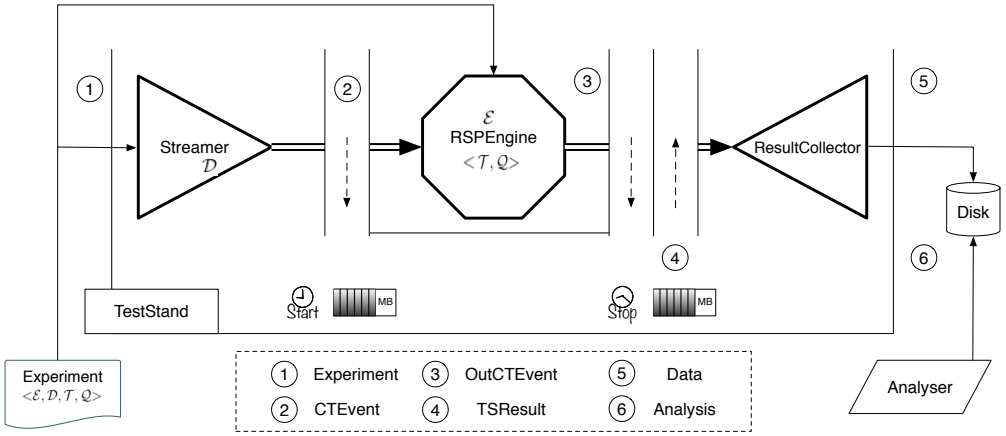


Figure 4.2 – The execution starts when the TEST STAND received the Experiment $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q} >$ as an input (1). The STREAMER pushes a CTEVENT, an RSP Stream $\mathcal{D}$ fragment, to the RSP Engine in in use $\mathcal{E}$ (2). In (3) $\mathcal{E}$ pushes to the RESULTCOLLECTOR and OUTCTEVENT with the results of the queries $\mathcal{Q}$ w.r.t the Ontology $\mathcal{T}$. The event is wrapped by the TESTSTAND into an TSRESULT event which contains also the measurement data sampled in (1) and (2) when $\mathcal{E}$ is not running. In (5) the data are persisted and then they are analysed in (6) by the ANALYSER.

of RDF triples with the same timestamp. In order to satisfy [R.12], it sends triple in N-Triple[1], which is the easiest RDF serialisation to parse.

In step (3), $\mathcal{E}$ pushes to the RESULT COLLECTOR an event OUTCTEVENT. It contains the current answer to the query $\mathcal{Q}$ registered in $\mathcal{E}$ given the ontology $\mathcal{T}$. The TEST STAND expects $\mathcal{E}$ to output result in N-Triple format.

Notably, to place any RSP engine on the TEST STAND (requirement [R.3]) $\mathcal{H}$eaven provides a simple software wrapper that, when it receives a CTEVENT, adapts it to the RSP engine specific format, pushes it in the RSP engine, and listens to the RSP engine output so to transform such an output in a OUTCTEVENT.

To measure performances (requirement [R.6]) the TEST STAND performs several actions both before step (2) and after step (3). Previous works about Stream reasoning [44] shows that the minimal performance measure set includes **Latency** – defined as the delay between the injection of an event in the RSP engine and its response to it –, **Memory Load** – defined as the differ-

[1] `http://www.w3.org/2001/sw/RDFCore/ntriples/`

ence between total system memory and the free one – and **Completeness & Soundness** of query-answering results. To measure latency, it starts a timer before (2) and stops it after (3). To measure memory load, it asks for the free memory of the system after step (3). Completeness & Soundness are evaluated with post-processing analysis of the query results data.

In step (4), those observations are added to the outputs of $\mathcal{E}$ as annotations and are pushed to the RESULT COLLECTOR. We name TSRESULT the event that contains the sensor data plus the query results produced by the engine.

The TEST STAND works in a single thread mode, blocking the execution of its components when it performs the measurements in (2) and (3) [R.4].

In step (5), the RESULT COLLECTOR saves the content of any TSRESULT [R.8] for post process analysis [R.9] executed through the ANALYZER.

4.2 Baselines

In Chapter 3, we state that a Systematic Comparative Research Approach needs initial terms of comparison to lead the investigation. $\mathcal{H}$eaven contains a set simple and easy-to-use RSP Engines called "Baseline". As the name lets to guess, these engines fulfil the four characteristics, detailed in Section 3.2, required by an RSP Engine to be classified as a baseline inside the SR research field. Thus, $\mathcal{H}$eaven Baselines are *Elementary*, *Relevant*, *Simple* and *Eligible*.

Early works on SR describe the most simple approach to create a stream reasoning system as pipelining a DSMS with a reasoner [49, 52]. The DSMS is responsible to handle the data stream, moving from infinite sequences to finite (and processable) sets of events. The reasoner instead applies SPARQL queries on this set of events, exploiting its reasoning capabilities over a context that can be considered as static, but remains continuous. We focus on RDF Stream Processing, whose foundations, as explained in Section 2.3, are:

1. RDF streams, detailed in Section 2.3.1,
2. A continuous extensions of SPARQLs, detailed in Section 2.3.2,
3. reasoning algorithms.

RSP Engine are those systems that can apply reasoning techniques upon rapidly changing information encoded in RDF (RDF Stream) and allow to

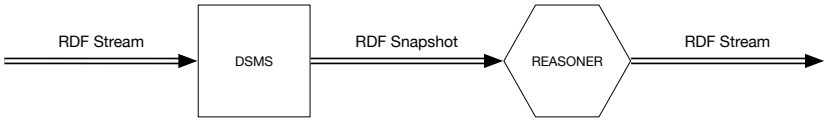
continuous querying on the data stream (Section 2.3.3). It is possible to develop an RSP Engine following the approach described above, which actually requires to develop the integration of two existing technologies, DSMS and reasoner, and to define how they can communicate. In the following, we describe how this design model fulfils the requirements we posed in Section 3.2.

Baselines *Elementarity* can be granted by choosing a DSMS which is a reliable solution in the Information Flow Processing context and a general purpose rule engine which is comparable with mature solutions.

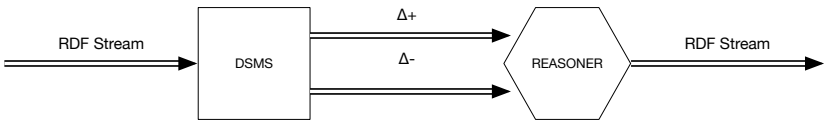
Elementarity is reached when the coupled elements are simple and valid terms of comparison w.r.t the state of the art.

Baselines *Relevance* requires to cover all the most important theoretical variants that the "pipeline approach" conveys. In terms of reasoning we can choose between two possible approaches and with reference to the data stream processing the choices are again two.

Four baseline implementations cover these two main design decisions about the RDF Stream Model and the Reasoning procedures.



(a) Baseline A: Naive



(b) Baseline B: Incremental

Figure 4.3 – The Architecture of $\mathcal{H}$eaven Baselines: (A) The Naive reasoning approach, the DSMS outputs an RDF Snapshot of the active windows at each time it slides. (B) The Incremental reasoning approach, the DSMS outputs a IRStream, the differences between the active window and the previous one: a Δ^+ represents the new incoming triples while Δ^- represents the outgoing ones.

The RDF Stream model describes how the input RDF Stream is processed, different systems accept data in different models, which depends on how RDF Stream is considered in terms of events contemporaneity. The two most relevant ones are:

- Triple-based model, where the events pushed in the DSMS are timestamped triples. The timestamps are non decreasing, i.e. different triples could have the same timestamp to denote that they are contemporary.

- RDF Graphs-based: the event pushed in the DSMS are timestamped RDF graph. The timestamps are increasing and the graph is used as a form of punctuation [47] to separate consequent portions of the RDF stream.

The Stream Reasoning depends on the way data flow from the DSMS to the reasoner. Figure 4.3 shows the two reasoning solutions related to the two triples data flow above:

- Naive solution: (Figure 4.3-A) the DSMS produces an RDF Snapshot of the active window. It sends the entire content of the window to the reasoner that materialises all the implied triples at each cycle. This is the approach implemented in the C-SPARQL Engine [10] and in Sparkwave [32].

- Incremental solution: (Figure 4.3-B) the DSMS outputs an IRStream, the differences between the active window and the previous one. The Δ^+ snapshot contains the triples that have just entered in the window, while the Δ^- snapshot contains the triples that have just exited from the window. The reasoner, using Δ^+ and Δ^-, incrementally maintains the materialisation over time. This approach is taken as term of comparison in [18] and it is inspired from [43].

Baseline Eligibility requires fair performance measurements w.r.t mature RSP Engines. The choice of the DSMS and the reasoner affect baselines Elementarily and it influences their performances too. To evaluate the Baselines, we must use the minimal measure set presented in Section 4.1: the latency, memory and the Completeness and Soundness of the query results. It is easy to compare latency and memory performance values, while for Completeness and Soundness further consideration are needed. To be fair with mature RSP Engine the baselines query results must output the correct answers under a given entailment regime. The recent work [19] explains the importance of external time control to ensure the RSP Engine output correctness (even

when overloaded). The Baselines must exploit the ability of some DSMS to be temporally controlled by an external agent, in order to ensure Completeness and Soundness of the results even in case of high stress condition. Comparable measure of latency and memory and external time control can prove the Baseline ELIGIBILITY.

Finally, baselines SIMPLICITY comes from the registered query $\mathcal{Q}$, and the entailment regime in use. $\mathcal{Q}$ should be eligible in terms of reasoning, which means having an high materialisation effort of the implicit information entailed by the content of the window, given the ontology chosen by the user. The entailment regime should be an RDF(S) fragment with a good trade off between complexity and the normative semantics and the core functionalities, for example ρDF (see Section 2.1.3).

4.3 Analyser

The ANALYSER design can be faced in two ways: (i) from an engineering viewpoint, it is composed by automatic tools that process experiment results to obtain human readable data; (ii) from a research point of view, it is a set of methods for data analysis which have the aim of hypothesis confirmation and of the improvement of theoretical models.

We follow the researcher vision (ii) and, thus, in this section define the methods that compose the analysis. Further details of the implemented tools, which sustain our investigation can be found in Section 5.4.

Figure 4.4 shows the different phases of the data processing. The methods that compose the ANALYSER can be divided into three main blocks, each one with different supporting tools and different goals.

The ANALYSER takes as input the raw data produced by the TEST STAND by executing the experiment, and the variables on which the analysis will be based on. The TEST STAND outputs raw data in times series format. The TEST STAND measurements set may vary according to requirement [R.7]. To this extent, the ANALYSER should be extendible too, and the variables of the analysis must be seen as an input provided by $\mathcal{H}$eaven user.

To properly compare results between n different RSP Engines data must be standardized. In the *Steady State Identification* block data are processed

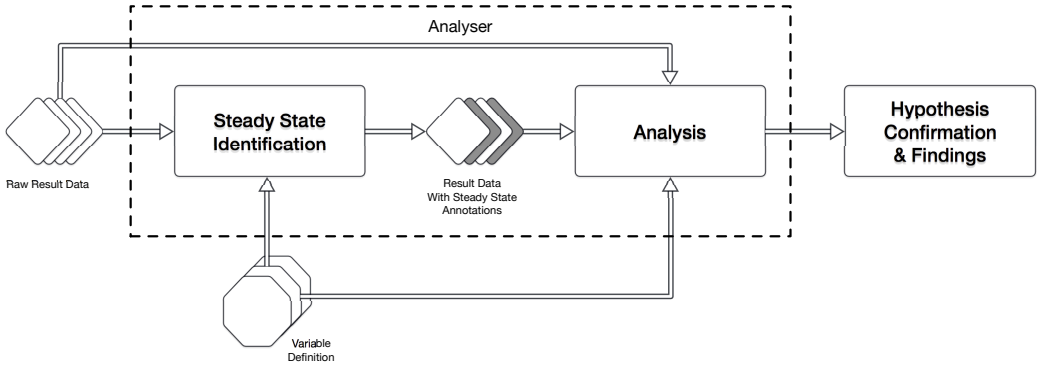


Figure 4.4 – Two inputs start the ANALYSER processing: the TEST STAND output and the investigation variables. At first, Steady State Identification (SSI) block indicates if a variable reached the Steady State. Analysis block (AB) builds the analysis exploiting both the initial inputs and the SSI block results. AS outputs an human readable data to answer hypothesis or state new insights upon.

according to the variables, identifying which variable has reached a Steady State condition.

The last step of the high level Analyser Block Schema, in the figure, consists in the formalisation of theoretical results. The aim of this step is obviously confirm or refute hypothesis formulated at experiment design level. However, $\mathcal{H}$eaven has the aim of sustaining the empirical research over RSP Engine, which allows a new kind of observation that may improve existing theoretical models of the Stream Reasoning area.

In the next sections we provided further details on the *Steady State Identification* block, Section 4.3.1, and about the *Analysis* block, Section 4.3.2.

4.3.1 Steady State Identification Block

The *Steady State Identification* (SSI) block is the first element in Figure 4.4. In general, it applies a pre-analysis of the raw data, evaluating the final condition of each variable and establishing if it has reached or not the Steady State condition. The Steady State is the moment when a dynamic system reaches the equilibrium for a certain variable. The SSI is a common step in almost any research on dynamic systems, because those systems usually have an initial transitory phase which inhibits generalisations and comparisons. Figure 4.5

shows the typical behaviour, in the time domain, for a certain variable and it also evidences the point when the series reaches the Steady State condition. The *Steady State Identification* block allows to understand the degree of reliability of the data, i.e. how we can assume a certain observation is confirmed and generalizable.

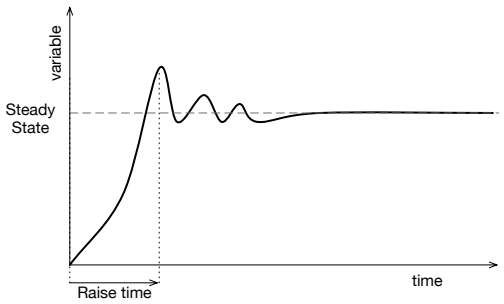


Figure 4.5 – Time Series Behaviour Example in Temporal Domain

4.3.2 Analysis Block

Once the Steady State is identified, it is possible to proceed with the central data analysis, which is summarised in Figure 4.4 by the *Analysis* block. This block exploits the *Steady State Identification* output to study the transitory phase, which is a crucial part of the dynamic system comprehension and, thus, of the Hypothesis confirmation.

The investigation can be decomposed in four levels with increasing details degree and different goals. Figure 4.7 is a graphical representation of the investigation stack implemented within the *Analysis* block, the detail level grows from the top to the bottom.
Before presenting the levels of the stack one by one, we introduce two concepts about the experiment analysis:

- *Intra Experiment Comparison* - it means building comparisons between variables upon a single, well-determined experiment.
- *Inter Experiment Comparison* - it means building comparisons of different experiments upon a single variable.

Level 0 - Dashboards

Dashboards are the highest level of analysis offered by $\mathcal{H}$eaven for *Inter-Experiment* comparison. Some statistical values like average (or maximum, minimum, median, etc.) are presented in a n-dimension radar plot, as the one in Figure 4.6, which involves all the variables selected during the experiment design phase. Visual comparison of the data through dashboards is natural when few variable are involved. It is easy to compare many solutions and identify which one is the best. The aim of dashboard is to compare experiments and pointing out the relation that occurs among the involved variables.

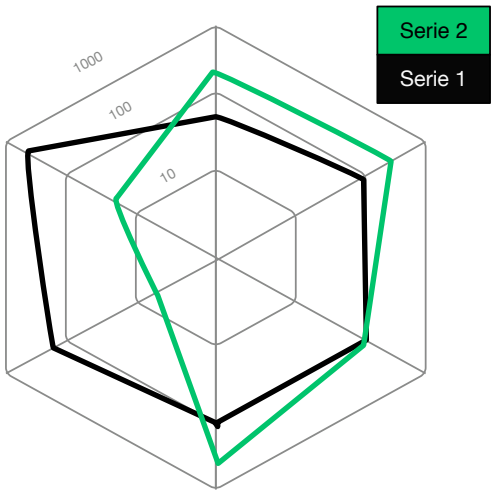


Figure 4.6 – Dashboard Example - Radar Plot

The idea of a single visualisation method, which allows to answer to any hypothesis, is desirable but not probable. Unfortunately, the reliability of the methods depend on the system complexity and not only on the complexity of the method itself. Thus, this level of analysis may not be able to represent the entire system complexity. The Steady State condition represents another point of weakness. If it is not reached by all the variable involved, the analysis generalisation can not be granted and dashboard relevance becomes frail. Further levels of analysis are required, at least for a better comprehension of those unpredictable results that refute even naive hypothesis, formulated on well known theoretical truths.

Level 1 - Statistical Values Comparison

This Analysis level focuses on a single variable at time (Latency, Memory etc.), in a certain statistical condition (Maximum, Minimum, Mean Value etc.) exploiting *Inter-Experiment* comparison.

To verify an hypothesis researches design and execute multiple experiments which variates for few parameters. This analysis is focused on the entries experiment set. The experiments must be arranged into an smart layout that highlights the differences between experiments upon one or more well-defined characteristics. The analysis involves one variable at a time, comparing the results over the experiment set.

The aim of Level 1 is identifying which parameter, if any, determines behaviour of the solution w.r.t the observed variable. $\mathcal{H}$eaven enables two possible analysis approaches within Level 1:

- *Quantitative* - The comparison results are present in percentage form, quantifying how much a solution is better than another one under some conditions.
- *Qualitative* - it is a simplification of the Quantitative approach. Sometimes we only need to understand which solution is the best, without focusing on numeric values. The Qualitative approach requires the definition of a tolerance threshold, for example 5%, to distinguish when a solution is better, worse or equal to another one.

Level 2 - Patter Identification

The comparison of a single statistical value over multiple variable (Level 0) or a single one (Level 1) may be insufficient to explain the RSP Engine behaviour. Starting from Level 2, visual analysis for *Inter-Experiment* comparison is introduced. As in Level 1, the investigation involves a single variable over the entire experiment set, disposed into an easy-to-read layout which points out experiment differences. Level 2 instead enables the comparison of the experiment set, presenting result data in a graphical way, which shows the system behaviour over all the experiment executions.

The aim of this level is highlighting if a certain variable follows one or more patterns among the experiment set. How to choose the correct graphical

representation depends on the variable nature and requires specific analysis. However, the most common ones for time-series are time domain form, value distribution or frequency domain representations. In general, Level 2 allows to visualise how the system behaves, which is not visible with a mere model investigation.

Level 3 - Visual Comparison

The Level 3 is the bottom level of the investigation stack. It focuses on a single solution at time and it exploits both *Inter-Experiment* comparison, with the aim to understand how different experiment execution are related, and *Intra-Experiment* comparison, with the goal of pointing out the relation between the variables over all the experiment. In the first case, Level 3 reproduces the Dashboard idea but over all the experiment execution. In the second case instead, it extends what done in Level 1 and 2, but focusing on a single visualisation at a time.

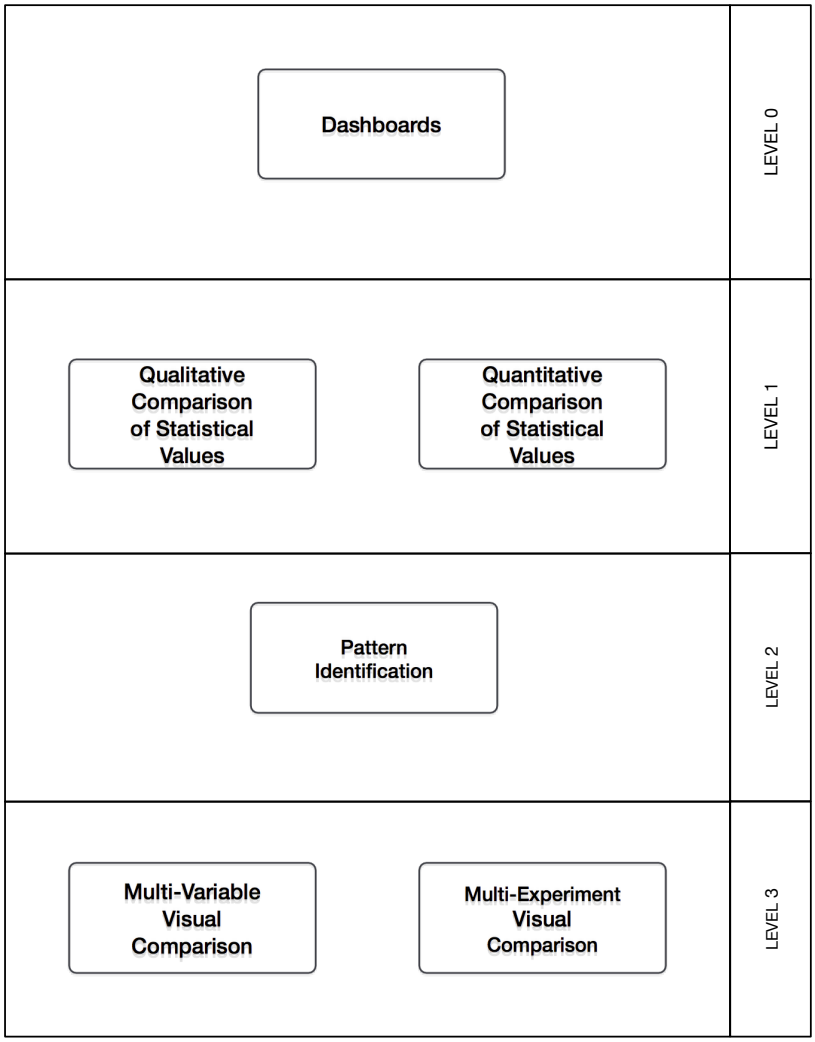


Figure 4.7 – ANALYSER Investigation Stack - At Level 0, all the variables are compared through statistical dashboard representation. At Level 1, statistical values of each single variable are globally compared. At Level 2, global graphical comparisons of the RSP Engine dynamics allow pattern identification. At Level 3 experiments (*Inter*) or properties (*Intra*) are compared one by one.

Chapter 5

Heaven - Implementation Experience

In Chapter 3, we posed the requirements that any solution must fulfil, in order to properly answer our research question. Then, in Chapter 4, we faced each requirement at design level, stating how an implementation of $\mathcal{H}$eaven must be realised to fulfil them.

In this chapter, we describe the implementation experience of $\mathcal{H}$eaven: Firstly in Section 5.1, we present the pillar concepts of this development: the basis abstractions, in Section 5.1.1, and the data model of the Test Stand, in Section 5.1.2. In Section 5.2, we introduce each $\mathcal{H}$eaven module: the Streamer, the ResultCollector and the Test Stand Supporting Structure. In Section 5.3, we present how we implement the four baseline RSP Engines. Finally, in Section 5.4, we describe how the Analyser Investigation Stack is realised w.r.t the Evaluation we will present in Chapter 6.

5.1 Test Stand

The architecture of the Test Stand consists of three stand alone modules that establish a mono-directional communication flow: the Streamer, the RSP Engine and the Result Collector. Moreover, the idea of an external structure which supports the other modules brings to the concept of the Test Stand Supporting Structure. All these components communicate exchanging events data, exploiting an event-based and modular architecture

as demanded by [R.10] and [R.11]. In the following, we presents the two abstractions that allow this architecture: *EventProcessor* and the *Event*. Then, we detail how the Test Stand Data Model is implemented.

5.1.1 Abstractions

Among all the requirements reported in Section 3.2, two of them are immediately relevant: [R.10], i.e. the need of an *Extendible Design*, and [R.11], which states the necessity of an *Event-base architecture* to properly face any RSPEngine. In order to to fulfil both, the Test Stand requires two main abstractions:

- The *Event* - which is required to build a hierarchical communication. Indeed, the Test Stand may handle three event flows: one internal to the RSP Engine module, one for the communication between modules and one to communicate with the user. Next section about data clarifies the communication structured.

- The *Event Processor* - guarantees the system to be modular and it standardizes the interaction simplifying the behaviour of each component in the system. Thus, a module is an *Event Processor* which can be positioned everywhere in the the Test Stand pipeline.

The requirement [R.4] directly influences the the Test Stand workflow stating that it must not be running when the RSP Engine is under execution. To cover this requirement, we designed each module as a Finite State Machine (FSM), which can work only specific states that allow processing (READY).

Each $\mathcal{H}$eaven module, the Baselines, and also for the Test Stand External Structure follows the FSM in Figure 5.1, implemented through two interfaces:

The *Startable* interface standardises two methods, *init()* and *close()* which allow to control the behaviour of the module at the beginning and the end of the execution. The interface allows to move from the CLOSED state to the READY through the *init()* method and from the READY to the CLOSED through the *close()* method.

EventProcessor interface completes the FSM schema, exposing the *process (Event e)* method. The method brings the module into the RUNNING state

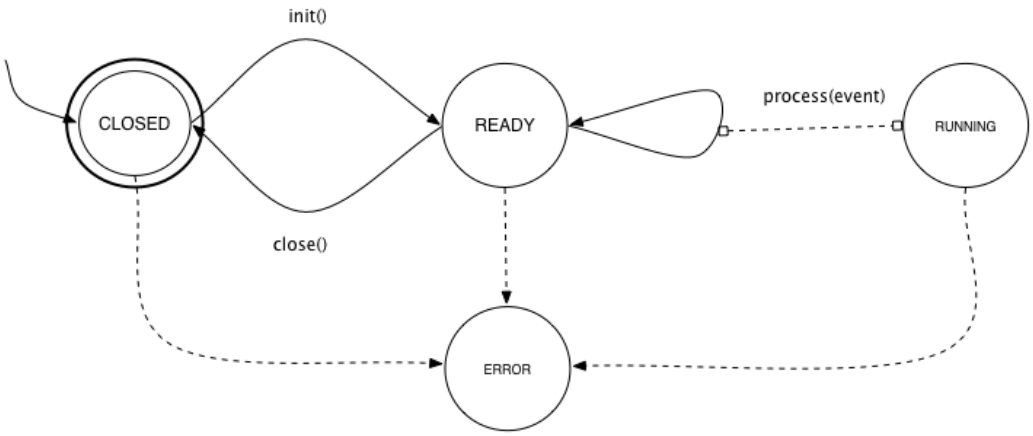


Figure 5.1 – The finite state machine diagram of any $\mathcal{H}$eaven module and the Baselines. It is realised through the *Startable* interfaces, which provide the *init()* and *close()* methods, and through the *EventProcessor* interface, which declares the *process(Event e)* method. The execution is possible only during the RUNNING state, which can be achieved by one and only module at time, as a constrain. The ERROR state prevents the propagation of erroneous behaviour to the data.

until the processing ends, and then back to the READY one. As a constrain, one and only one module can be in the RUNNING state in a certain moment during the execution.

Finally, the ERROR State, which can be reached from any point of the execution, prevents the propagation of errors over result data: when a module fails the execution is stopped without saving the erroneous data (last event) and reporting the error to the user.

[R.4] is fulfilled because each module in the systems fulfils [R.10] and [R.11]. $\mathcal{H}$eaven Modules can be explicitly controlled by the Test Stand Supporting Structure, which starts and stops the processing through the methods exposed by the *Startable* and *EventProcessor* interfaces. It bring the modules from one status to another one, according with the FSM schema in Figure 5.1.

5.1.2 Data Model

In the previous sections, we stated that the TEST STAND and its modules exploit an event-based communication, as required by [R.11]. Chapter 4 describes $\mathcal{H}$eaven workflow and how it exchanges events during the execution. Each event represents different data, depending on its position in the workflow. The *Event* concept we introduced, which is implemented as an interface, allows to give a hierarchy to the exchanged events. In general, the Test Stand handles four kinds of event, which are reported in Figure 5.2 and defined as follow:

- *Experiment* - it represents the tuple $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q} >$, indicating which RSP Engine ($\mathcal{E}$) will be tested and with which queries ($\mathcal{Q}$), data ($\mathcal{D}$) and ontology ($\mathcal{T}$) will be used. It contains the execution start time and the end time. A specific field indicates if the current implementation of the engine exploit external timing, while the *type* parameter indicates which kind of testing will be applied (SOAK o Stress for example).

- *CTEvent* - it contains a set of contemporary triples, wrapped in the *TripleContainer*. The id field identifies is unique for the event within the experiment.

- *OutCTEvent* - it represents the event produced by the RSPEngine after processing the active window. Figure 5.2 shows the inheritance relation with *CTEvent*: *OutCTEvent* extends the *CTEvent* adding the output-Timestamp field.

- *TSResult* - it wraps the *OutCTEvent* adding the information about the measurements data gathered during the execution: memory, sampled ante e post processing, and latency. Two boolean fields allow to mark complete and soundness result, if it is evaluated at runtime (see Section 3.2)

$\mathcal{H}$eaven requires an initialization phase to prepare and input the *Experiment* into the TEST STAND. The current implementation exploits a property file with the Experiment parameters: ID and the tuple $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q} >$.

The *CTEvent* and the *OutCTEvent* contain RDF triples in NT-Triple[1], which is the easiest RDF serialisation to parse. This serialisation was chosen to

[1] http://www.w3.org/2001/sw/RDFCore/ntriples/

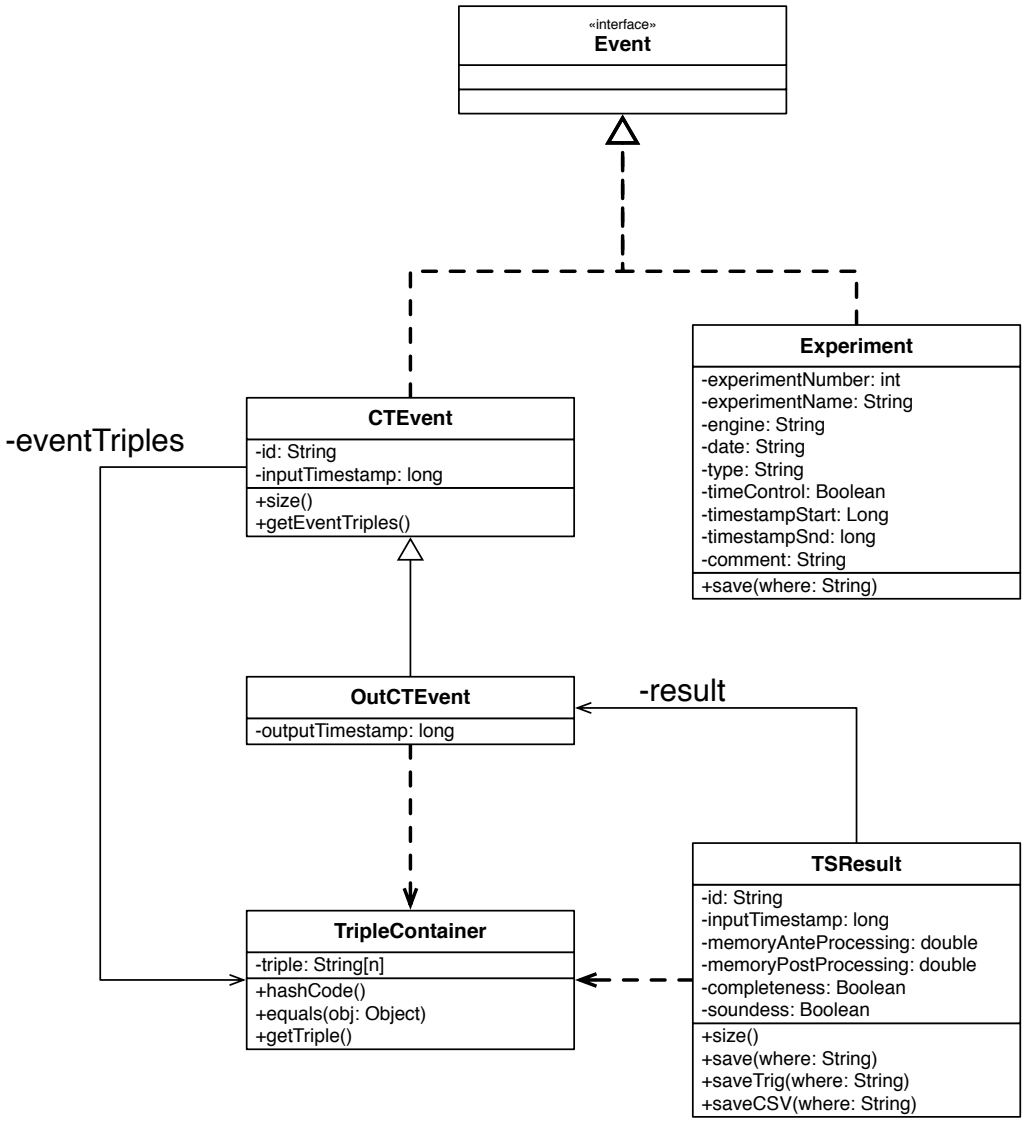


Figure 5.2 – The figure shows the events that $\mathcal{H}$eaven Test Stand and its modules exchange during the execution of an experiment. All of them implement the *Event* interfaces that is at the top of the class hierarchy.

fulfil requirement [R.12], which demands an *Easy-to-Parse RDF Serialisation for the events presented to the RSP Engine in exam.* Figure 5.2 shows also that the RDF Triples are stored in the events into the *TripleContainer* wrapper: we redefine the triple hashcode and equals method guaranteeing their uniqueness within an *CTEvent* or *OutCTEvent*.

5.2 Test Stand - Modules

In this section we present the two of modules which compose $\mathcal{H}$eaven, the Streamer, in Section 5.2.1 and the ResultCollector in Section 5.2.2, while the RSPEngine will be introduced in Section 5.3. Moreover, in Section 5.2.3 we details the Test Stand Supporting Structure.

In Section 5.1.1, we define a module as an *EventProcessor* which can be positioned everywhere in the the Test Stand pipeline. We state that a module must implements the *Startable* interface, which completes the FSM schema in Figure 5.1 with the *init()* and *close()* methods.

5.2.1 Streamer

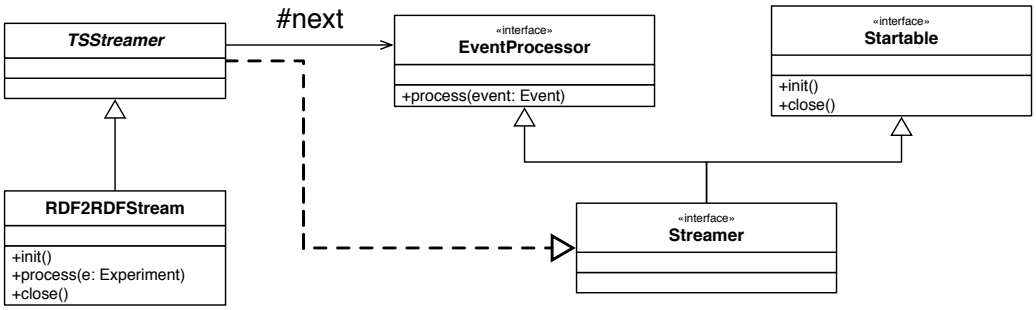


Figure 5.3 – *RDF2RDFStream* Streamer Implementation. The *RDF2RDFStream* extends the *TSStreamer* abstract class, which defines the module as a *Startable EventProcessor* of *Experiment*s

Figure 5.3 shows the current implementation of the *Streamer* interface, which is the head of the Test Stand pipeline. Actually, the *Streamer* is implemented as the *TSStreamer* abstract class, which is specialised into processing of *Experiment* events.

The *Experiment* events are externally instantiated and then *TSStreamer* receives them. It can start the processing only if it is initialised. It communicates with another *EventProcessor*, called *next*, which can receive *CTEvent*s. In Figure 5.3 the *next* is represented by the labelled arrow. Notice that, also the *next* must be initialised before starting the communication, otherwise the ERROR state will be reached, since the processing is not allowed.

Figure 5.3 contains also the *RDF2RDFStream* implementation, whose internal structure can be seen in Figure 5.4. The *RDF2RDFStream* was developed to conduct experiments as they are presented in Chapter 6. We use the LUBM Benchmark to generate the data for the experiments, but it actually generates static data and, thus, we translate them to a streaming scenario through the *RDF2RDFStream*, which builds an RDF Stream attaching a timestamp to the static data produced by LUBM.

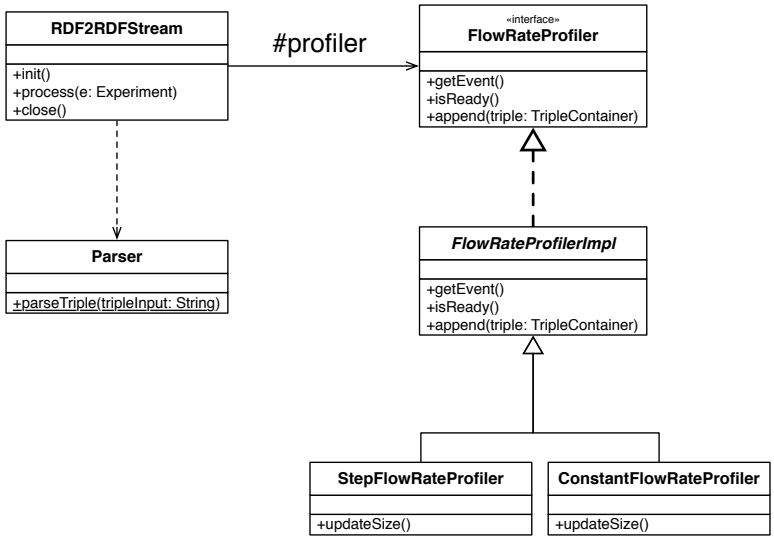
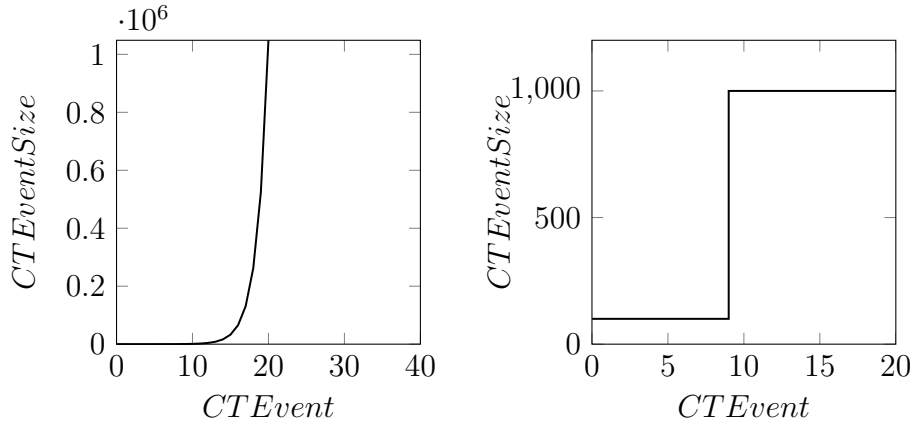


Figure 5.4 – The *RDF2RDFStream* exploits two subcomponents to build an RDF Stream: the *Parser*, which allows to read in memory RDF Triples, and the *FlowRateProfiler*, which attaches to RDF triple a timestamp and it builds *CTEvent*s according to a function f. The *FlowRateProfileImpl* provides implementation of the *append(TripleContainer tc)* method and of the two utility methods *isReady()* and *getEvent()*. Specific implementations of the *FlowRateProfiler* control the function f through the *updateSize()* method.

The *Parser* component, in Figure 5.4, can be accessed statically. It reads in memory, one by one the triples, in the file, guaranteeing data independence [R.1] and it does not influence the memory footprint [R.5], because it allocates only the memory necessary to parse a triple.

Figure 5.4 also includes the *FlowRateProfiler*. This component determines the number of triples to add to a *CTEvent* and it builds such an event. In this way, *RDF2RDFStream* can generate many RDF Streams to use as $\mathcal{D}$, which differ on the number of contemporary triples.



(a) Exponential Growing Size: $y = 2^x$ (b) Step Growing Size with K1=100 and K2=1000 after 9 CTEvents

Figure 5.5 – The FlowRateProfiler is able to calculate CTEvent size according to a function which relates the number of triple to the CTEvent id.

The *FlowRateProfilerImpl* implements *FlowRateProfiler* interface. It provides a common implementation for the method *append(TripleContainer tc)*, which adds to the current *CTEvent* an RDF triple, and of the two utility methods *isReady()* and *getEvent()*. What variates between different implementations of this component, is the *updateSize()* method.

The *FlowRateProfiler* creates *CTEvent* according to a function $y = f(x)$, in which x is the *CTEvent* id and it results that y is the number of triple this *CTEvent* will contain. f is implemented within the *updateSize()* method. For example if we decide to increase linearly the number of triples inside a *CTEvent* the function f will be:

$$y = x, \text{ where } x, y \in N$$

The first event (E0) will contain zero triple, E1 will contain only one triple, E4 will contain four triples and so forth. Another possibility is to increase exponentially the number of triples inside a *CTEvent* :

$$y = 2^x, \text{ where } x, y \in N$$

The first event (E0) will contain one triple, E1 will contain two triple, E3 will contain eight triples and so forth. Figure 5.5.a shows the resulting behaviour plotting the triple number on y-axis and *CTEvent* id on x-axis.

In the current stage of development, we include four implementations of the *FlowRateProfiler*, two of them are related to our experiments:

- *ConstantFlowRateProfiler* - it maintains the same number of triples for each event over all the experiment:

$$y = K, \text{ where } K \in N$$

- *StepFlowRateProfiler* - it maintains a constant number of triple $K1$ inside a *CTEvent* for x occurrences, then it suddenly changes the number of triple y form $K1$ to $K2$ where $K2 >> K1$. The number of *CTEvents* x is specified in the set-up phase of the component. Figure 5.5.b contains the resulting plot of implemented function which follows:

$$y = \begin{cases} K1, & \text{if } x < M \text{ where } K1, M \in N \\ K2, & \text{if } x >= M \text{ where } K2 >> K1, K2 \in N \end{cases}$$

The remaining two *FlowRateProfiler* implementations in $\mathcal{H}$eaven that are not related to our evaluation are:

- *LinearStepFlowRateProfile* - it streams x CTEvents of dimension y, in terms of triples, then linearly increase the number of a quantity M:

$$y = x * M, \text{ where } x, y, M \in N$$

- *PoissonFlowRateProfiler* - it changes y and x according with a parameter λ, which indicates the Expected Value of a Poisson Distribution that is

$$y = e^{-\lambda \frac{\lambda^n}{n}}, \text{ where } n \in N$$

5.2.2 Result Collector

The ResultCollector is the data acquisition system that receives and persists the query results and the measurements data gathered by the Test Stand during the execution of an experiment.

The UML Schema, in Figure 5.6, shows the *ResultCollector* interface, another proxy for the *EventProcessor* and the *Startable* ones. The current implementation is the *TSResultCollector*, which stays at the end of the Test

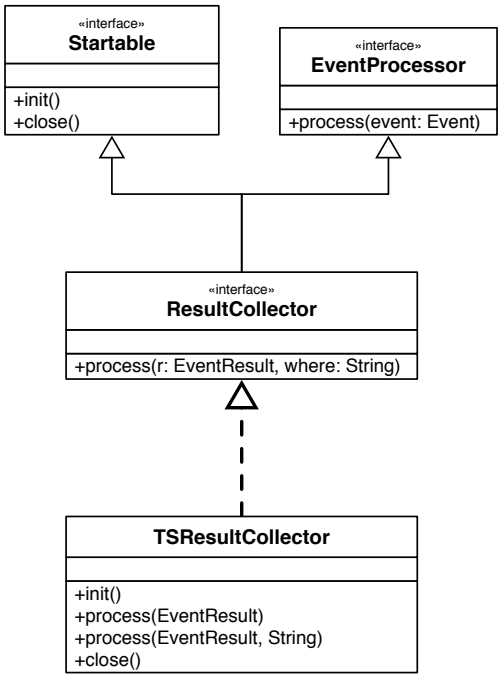


Figure 5.6 – The current implementation of the ResultCollector interface is the *TSResultCollector* which processes events that implement the *EventResult Interface*. *ResultEvent* interface hides the saving procedure, delegating the implementation to the event provider through the *save()* method. *TSResultCollector* exposes also the method *process(Event e, String where)*, which allows the caller to specify the destination.

Stand pipeline. The *ResultCollector* is responsible of saving data independently from their format, since requirement [R.7] demands to *enable users extensions to the basic measurement set*. The *TSResultCollector* applies a general saving procedure exploiting the *EventResult* interface, which exposes the *save(String where)* method, delegating the implementation of such procedure to the provider of the event. Figure 5.6 shows the relation between the *EventResult* interface and the *TSResultCollector*, which specialises the processing method to *process(EventResult)* and it knowns the general destination of the data, but it also exposes a secondary one *process(EventResult er, String where)*, which allows the caller to specify the saving path.

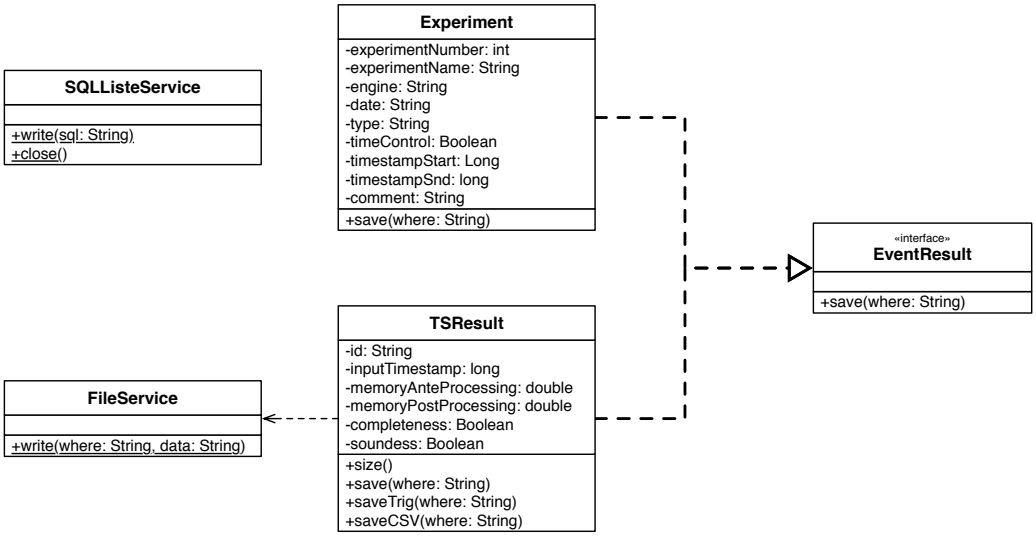


Figure 5.7 – The *Experiment*, the *TSResult* and the *OutCTEvent* implement the *EventResult* interface, hiding the saving procedure through the *save(String where)* method. In the current implementation, they exploit the *FileService* and the *SQLLIsteService* classes, to avoid concurrent accesses to the file system.

Figure 5.7 shows which events in the system exploit the *EventResult* interface. In the current implementation, the *TSResultCollector* handles two kinds of event:

- *TSResult* - it saves the data of the query results into a TriG[2] file where the graph name is the event id inside the experiment, while it saves the measurements data into a CSV[3] file that represents the time series w.r.t events id.

- *Experiment* - it saves the experiment metadata and the tuple $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q} >$ collapsed into a generic description field into SQLite[4] database.

Both saving procedures exploit a service class, respectively the *FileService* and the *SQLLIsteService*. In Figure 5.7, we describe such services, which expose static methods to interact with the file-system. The goal is reduce system complexity and avoid parallel interactions that may influence the experiment, offering a single point to interact with the file-system.

[2]http://www.w3.org/TR/trig/
[3]$http://en.wikipedia.org/wiki/Comma-separated_values$
[4]https://sqlite.org/

5.2.3 Test Stand Supporting Structure

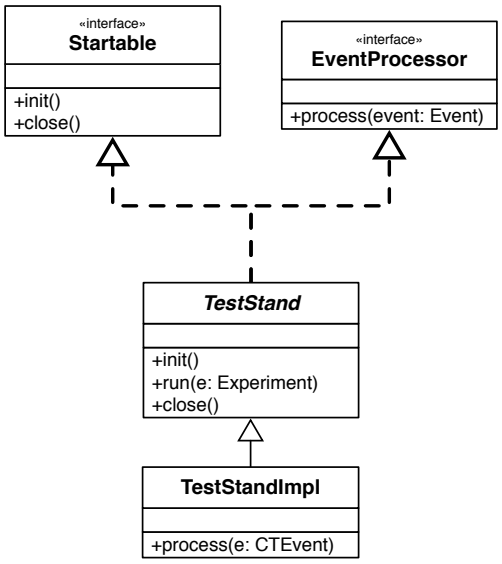


Figure 5.8 – The *EventProcessor* and the *Startable* method are implemented in two class levels: the *init()* and *close()* methods depend on the *TestStand* abstract class, while the *process(CTEvent e)* method is implemented by *TestStandImpl* class, which represents the Test Stand Supporting Structure. Moreover, it exposes also the *run(Experiment e)* method to start the processing.

$\mathcal{H}$eaven Test Stand was defined as set of modules which interact exchanging events during the execution. However, Chapter 4 describes at the design level the presence of an supporting structure which orchestrates the communication between the Streamer, the RSP Engine and the ResultCollector. This supporting structure also exposes the APIs for users interaction. Figure 5.8 represents this idea into an UML schema where the abstract class *TestStand* implements both the *Startable* interface, defining the *init()* and *close()* methods and also the *EventProcessor* interface, however it leaves the implementation of the *process(CTEvent e)* method to the current implementation: the *TestStandImpl*.

The relation between the *TestStand* and other modules is presented in Figure 5.9. The *TSStreamer*, the *RSPEngine* and the *TSResulCollector* are linked to the *TestStandImpl* through an initialisation class which receives the configuration file, and sets up these modules according with the requirements [R.1] [R.2] and [R.3] (respectively data independence, engine independence and

query independence). Once the set-up phase is completed, the *TestStandImpl* is initialized and it consequently initialises all the upstanding modules. The *Experiment* is created externally and the *TestStandImpl* receives it to start the execution.

During the execution, *TestStandImpl* gets the *CTEvents* from the *TSStreamer* and it sends them to the *RSPEngine* (see Section 4.1.3). The *TestStandImpl* gathers the measurements data according with the *Experiment* specification. It calculates latency starting a timer when the *CTEvent* arrives and stopping the timer when the *RSPEngine* outputs the results as *OutCTEvent*. It retrieves the memory usage asking the JVM in both the points above [R.6]. To fulfil requirements [R.7] any new measurement can take place only when the *RSPEngine* is not running. Once the *OutCTEvents* comes form the *RSPEngine*, the *TestStandImpl* immediately wraps the event into a *TSResult*, which is sent to the *TSResultCollector* to persist the query results and the measurement data, fulfilling [R.8] and supporting [R.9] for further analysis with the Analyser.

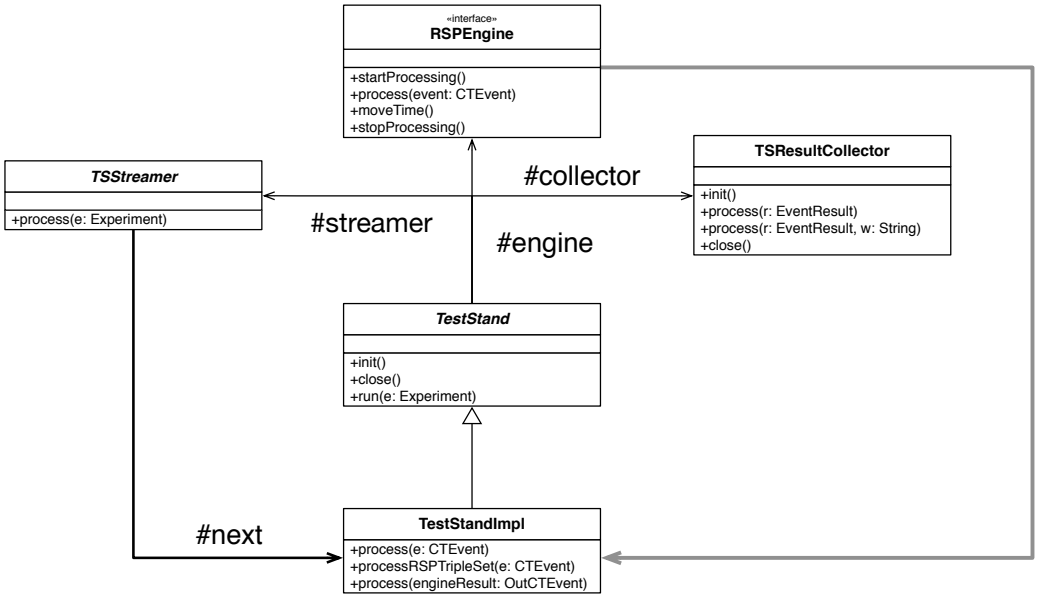


Figure 5.9 – The *TestStand* contains references of the *TSStreamer*, the engine behind the *RSPEngine* interface and the *TSResultCollector*. The *TSStreamer* and current *RSPEngine* are linked to the next element in the pipeline through the *TestStand*. Two arrows, labelled with "next", point to the *TestStand* indicating that it receives the events from all modules and it orchestrates the communication. The arrow that starts from the *RSPEngine* is coloured in gray to highlight that it can not be seen at this level of detail, because *RSPEngine* is an interface. The *ResultCollector* receives the result to save at the end of each cycle and, when it returns the call, the process ends.

5.3 Baselines

$\mathcal{H}$eaven Baselines are four elementary implementations of an RSP Engine, which cover the requirements from [R.13] to [R.16] and implement the pipeline of a DSMS with a reasoner following the proposal presented in Section 4.2. The RSP Engine pipeline is composed by Esper[5], a mature commercial DSMS, with the Jena general purpose rule engine[6], a flexible reasoning engine. The aim of the choice of Esper and Jena is fulfilling requirement [R.14], baselines Eligibility by coupling two mature solutions for stream processing and reasoning and, thus, obtaining a fair solution in the SR context. Moreover, both Esper and Jena are open source solution, which allow us to release $\mathcal{H}$eaven as open source too.

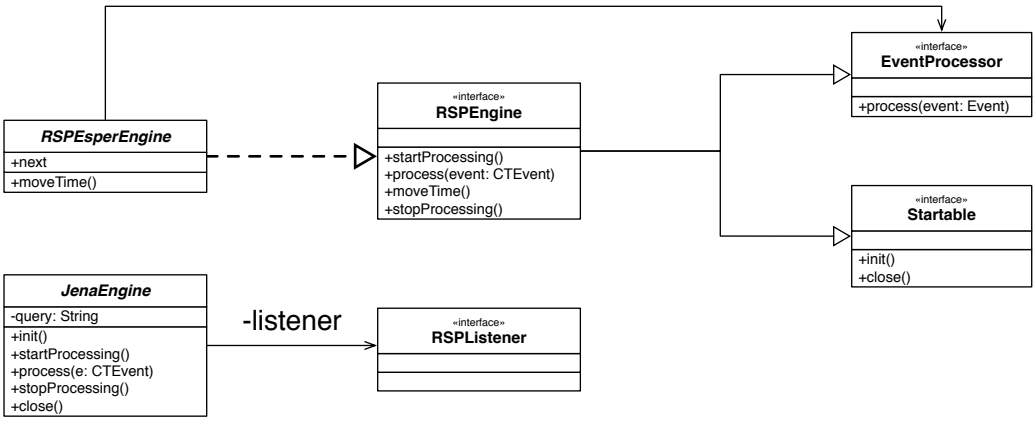


Figure 5.10 – The *RSPEngine* interface extends both the *EventProcessor* and the *Startable* one. The *RSPEsperEngine* class implements the interface adding the Esper runtime to handle its internal events. The *JenaEngine* together with the *RSPListener* integrate the Jena reasoning system into the engine.

Figure 5.10 presents how the baselines are implemented. The general structure exploits the *RSPEngine* interface, a proxy for both the *EventProcessor* and the *Startable* interfaces described in Section 5.1.1.

$\mathcal{H}$eaven Baselines integrate Esper as the DSMS, which composes the first half of the RSP Engine pipeline. The *RSPEsperEngine* abstract class implements the *RSPEngine* interface in order to share the Esper runtime definition for all the Baselines. They exploit the ability of Esper to be temporally con-

[5] *http* : *//www.espertech.com/esper/*

[6] http://jena.apache.org/documentation/inference/#rules

trolled by an external agent[7]. Thus, the internal time flow can be synchronised by sending time-keeping events. In this way, it possible to ensure the complete and soundness of query results, even in case of high traffic load. To enable external time control the *RSPEngine* interface exposes the *moveTime()* method. The *RSPEsperEngine* implements *moveTime()* encapsulating the logic to send a time-keeping event into Esper: one time-keeping event is sent before injecting the triples within a CTEvent and the next one after all triples in CTEvent were sent. In this way, all the triples in the CTEvent are considered contemporary by the Baselines.

The RSP Engine is in the middle of the Test Stand pipeline and, thus, it is has to communicate with the following module. The *RSPEsperEngine* has a reference to a general *EventProcessor*, represented in Figure 5.10 by the arrow labelled as "next", which can be any module which processes *CTEvent*. In the current implementation, the Test Stand Supporting Structure, implemented as the *TestStandImpl* class, follows a *RSP Engine* to intercept the outcoming *OUTCTEvents*.

We draw, in Figure 5.10, the UML schema of the Baseline implementation. The *JenaEngine* abstract class links the DSMS to the reasoner, an thus it requires a further component, the RSPListener, to complete the RSP Engine pipeline and realise the system we describe in Section 4.2

The reasoner stage is realised as shown in Figure 5.11. Different implementations of the listener, which all belong to the *RSPListener* interface, variate the reasoning approaches between Naive or Incremental. The *JenaNaiveListener* or the *JenaIncrementalListener* partially fulfil requirement [R.15] (baseline relevance) in terms of reasoning. None of the two specifies the entailment regime and the TBox, which must be defined with specific implementations like the *JenaRHODFNaiveListener*, as it is visible again in the Figure 5.11.

The Baselines relevance, demanded by [R.15], is only partially fulfilled by the alternative reasoning approaches. It comes also from the different implementations of the RDF Stream model, graph based or triple based. Esper runtime demands to registers the events that it has to handle. Figure 5.12 shows how events are implemented: they belongs to the *JenaEsperEvent* interface, which exposes three methods used by the *RSPListener* to manage the

[7] `http://esper.sourceforge.net/esper-0.7.5/doc/reference/en/html_single/index.html#api-controlling-time`

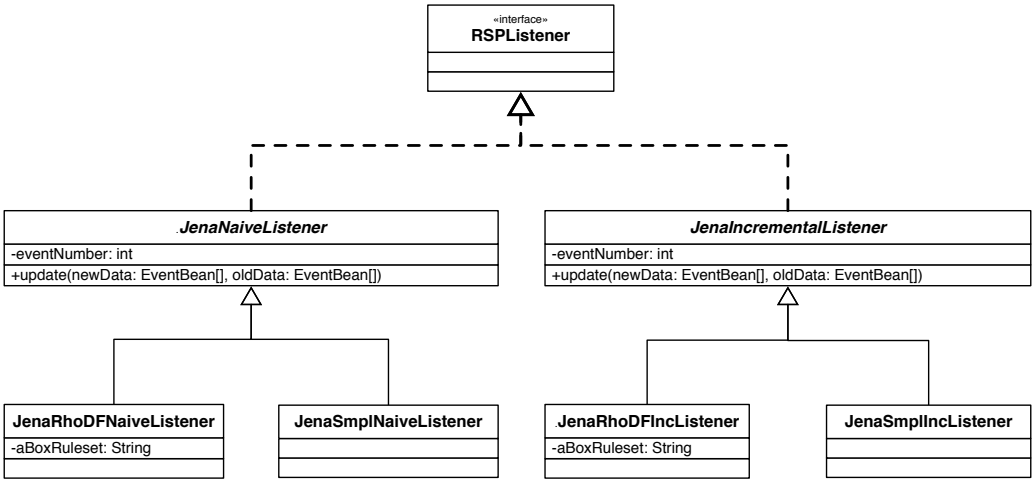


Figure 5.11 – The *RSPListener* is actually a proxy for the native Esper *UpdateListener*. We implement the interface following two possible reasoning approaches, Naive and Incremental, respectively into the *JenaNaiveListener* and the *JenaIncrementalListener*. Further implementations like the *JenaNaiveRhoDFListener* allows to specify to the listeners the entailment regime and the TBox.

active window independently from the event implementation. The methods *addTo(Graph g)* and *removeFrom(Graph g)* delegate to the event the inserting and deleting operations, in a transparent way for the *RSPListener*; the method *serialise()* unrolls the event into a set of statements, in this way the RSP Engine can output an *OutCTEvent* independently from the RDF Stream implementation: *GraphEvent* or *TripleEvent*

When a *CTEvents* comes to the RSP Engine, it will be transformed into the events handled by the DSMS, contained in Figure 5.12. This translation process influences the latency calculus, because the time spent by the engine to translate events from the RDF Stream into its internal mechanism may be relevant. Once the processing is complete, the output of the RSP Engine is translated again into an *OutCTEvent* and passed to the next *EventProcessor*, again spending time that influences the latency.

This assumption is inspired by the related work [12], which endorses the idea of black box testing provided by $\mathcal{H}$eaven. [12] states that every buffering or translation process, which is applied by the engine in use, must be considered part of its internal mechanism and, thus, part of the performance measurement.

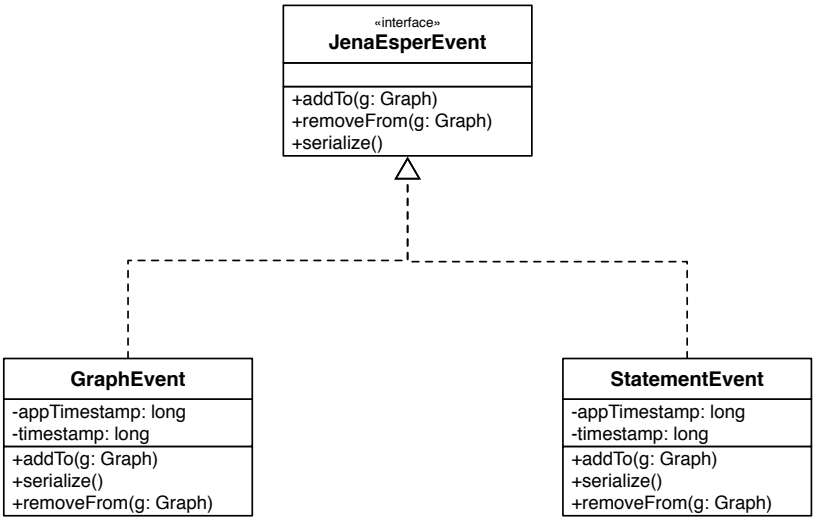


Figure 5.12 – All the event registered to Esper belong to the *JenaEsperEvent* interface, which exposes methods to handle the reasoning independently from the RDF Stream implementation: *GraphEvent* or *TripleEvent*. The triples received by the *RSPEngine* can be pushed into Esper as a complete graph or as a set statements. To handle the active window, independently from the event implementation, the interface exposes the method *addTo(Graph g)* and *removeFrom(Graph g)*, while the *serialise()* methods unroll the event into a set of statements, in order to build an outgoing *OutCTEvent*.

The current Baseline implementations divide the different architectural elements and delegate to each element a specific task to share the majority of the code and thus fulfilling [R.16], which demands baseline Simplicity.

5.4 Analyser

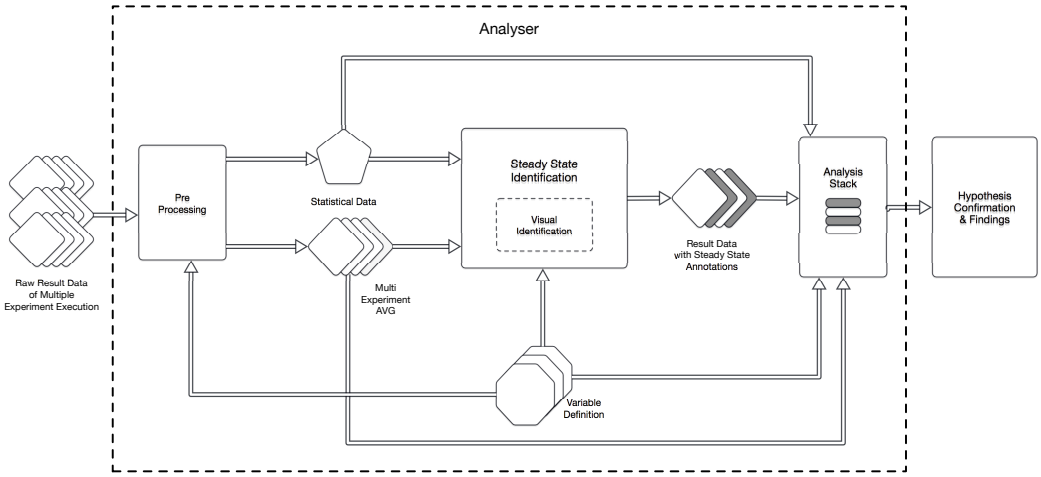


Figure 5.13 – The ANALYSER block schema in figure extends the one in Figure 4.4, with several implementation details. In particular, a new initial block, named *Pre-Processing*, operates before the other ones on the experiment raw data: it averages among multiple executions of the same experiment, to obtain a single reliable dataset; it calculates statistical relevant values (maximum, minimum, median etc) for each experiment w.r.t the involved variables that it receives as input. The *Pre-Processing* outputs are consumed by both the *Steady State Identification* block and the *Analysis* block.

In this section, we introduce which analysis tools sustain each level in the investigation stack described in Section 4.3 and how they are realized in the current implementation of $\mathcal{H}$eaven.

The relation between the hypothesis and the tools that sustain the analysis is deep and, thus, it is hard to generalise the investigation toolset. The Hypothesis depend on the the research question, while the tools are related to the nature of the data and to the experiment. However, there are several genera characteristics, which are meaningful independently from both the hypothesis and experiment. Thus, we can develop a basic toolset which sustains the entire investigation stack presented in Section 4.3.

Figure 5.13 shows the different phases of data processing. It refers to the original block schema presented in Chapter 4, but Figure 5.13 provides further implementation details.

The Figure 5.13 shows the ANALYSER receives two inputs:

- the raw data form the experiments,
- the variables to build the analysis.

In the original block schema, (see Figure 4.4) both inputs directly enter the *Steady State Identification* block (SSI) and the *Analysis* block (AB). In Figure 5.13 instead, the first block in the process is the *Pre-Processing* block. Empirical analysis can not rely on a single execution of an experiment, because even if the *Test Stand* is designed to be system independent, we do not have the complete control of the active processes within the execution environment. Strange behaviours may happen while an experiment is running. In order to reduce, and possibly eliminate, the outliers, multiple runs of the same experiment must be mediated obtaining the average measures. The *Pre-Processing* block ensures data reliability extrapolating a unique dataset from multiple executions. Moreover, time series describes how a dynamic system evolves over time, so it is meaningful to attempt hypothesis verification through statistical values, which always consider the the Steady State to allow the generalisation of the insights. The *Pre-Processing* block calculates most common statistical metrics as average, standard deviation and maximum or minimum for a certain variable.

Once we have reliable data, the *Steady State Identification* block and the *Analysis* block cans start the analysis process on the input variables and the *Pre-Processing* outputs.

Finally, researches can read the analysis point out insight and theoretical results as the last block in the process describes.

The following sections contain further details about the *Steady State Identification* block implementation, Section 5.4.1, and about the *Analyser* block with the investigation stack, Section 5.4.2.

5.4.1 Steady State Identification Block

The Steady State Identification block aims of determining if a a certain variable has reached the Steady State condition (see Section 4.3.2). Automatic procedures to identify the State State condition exist, but they require dedicated studies, which will be faced as future works. Currently, the SSI block is not

automated. It exploits data visualisation techniques, to identify if and when a variable reaches the Steady State condition. We plot each variable trend in the time domain over all the experiment execution. Then, we manually exclude the initial warm-up phase from the data evaluation.

We know that the graphical method is limited, because it must be applied for each system variable, and human criteria cannot be reliable in this kind of analysis as automatic procedures, which exploit tested algorithms. Moreover, different variables may reach the equilibrium at different times, so it is researcher responsibility to properly identify the different conditions for each variable involved. Thus, we include this further development as future work.

5.4.2 Analysis Block

The ANALYSER block includes five analysis levels, (see Section 4.3 with increasing degrees of detail, and where the comparative research approach is declined either via visual analysis or statistical investigation. The graphical analysis method is more qualitative then the statistical one, but reading the information presented in graphical way can be preferred in those cases where numerical data are not so meaningful. On the other hand, the statistical investigation method demands more complex instruments to obtain the data, but they allow to answer also more elaborate questions. In the following, we present the current implementation of all the analysis levels.

Level 0 - Dashboards

Figure 5.14 contains an example of the possible Dashboard representations. We implement the dashboard to represent data on a bi-dimensional Cartesian space where memory and latency are the axis of the graph. This kind of representation allows to define solution dominance, w.r.t the involved variable, through inter-experiment comparisons. Thus, we can easily state which RSP Engine, if any, is better then another.

Level 1 - Statistical Values Comparison

Tables 5.1 (a) and (b) show two examples of statistical investigation. Table 5.1.a contains the qualitative comparison of two solution over a given variable, while Table 5.1.b offers a deeper detail level, the quantitative comparison,

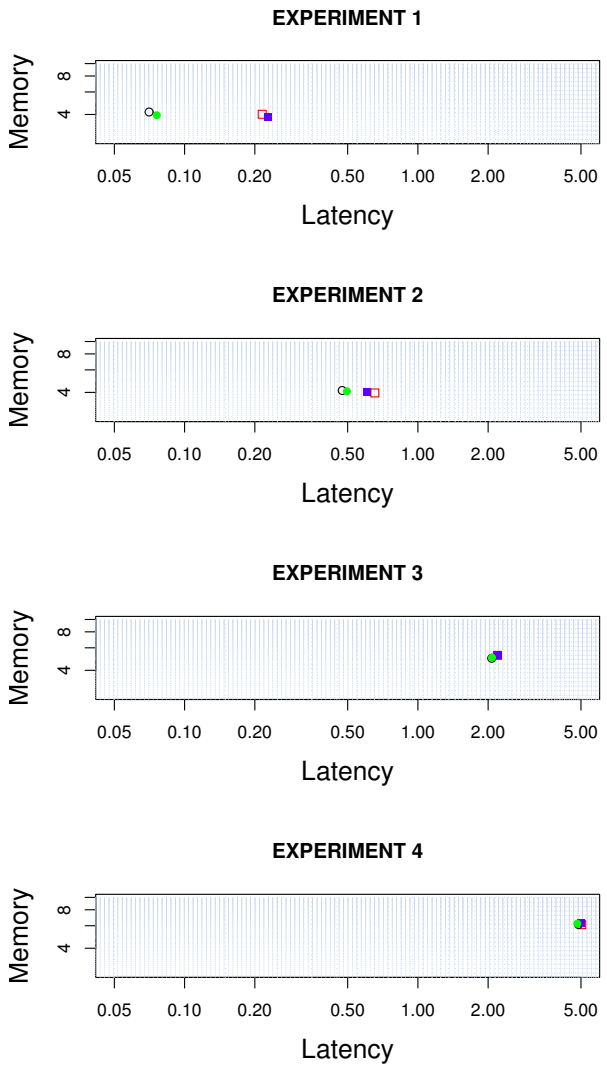


Figure 5.14 – ANALYSER Investigation Stack - Level 0 - Dashboard Representation Examples.

showing how much a solution is better than an other. How to choose the proper level depends on the needs of the research.

(a) Symbolic Comparison of variables A vs B on Experiment 1

A vs B Comparison	Experiment 1 Condition A			
	≃			
Experiment 1	A	≃	A	B
Condition	A	≃	≃	B
B	A	≃	B	A

(b) Symbolic Comparison of variables A vs B on Experiment 1

A vs B Comparison	Experiment 1 Condition A			
	≃			
Experiment 1	10%	≃	42%	33%
Condition	23%	≃	≃	12%
B	20%	≃	22%	22%

Table 5.1 – Analyser Investigation Stack - Level 1 - Example of qualitative-comparison over two variables (a) and quantitative-comparison over a common variable (b).

Table layout is a key-point for Level 1 representations. Table axes represent the variation of two different experiment properties A and B. Different experiments influence the behaviour of an RSP Engine in different ways, and Level 1 allows to point out this differences with *Inter-Experiment* comparisons. Thus, we can move on the horizontal axis of Table 5.1.a, which means variate the Condition A, to appreciate those differences.

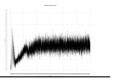
Actually this kind of analysis is possible thanks to a report, which contains all the meaningful statistical values for the experiments. The report can be further manipulated to obtain the table visualisation.

Level 2 - Patter Identification

Level 2 exploits the same experiment layout of Level 1 to compare many graphical representation of the experiment variable. Two examples of memory analysis at Level 2 are reported in Tables 5.2 (a) and (b). Table 5.2.a show the memory behaviour in time domain. It allows to answer questions like "How does the system change the memory behaviour modifying the input dimension?". Table 5.2.b reports the memory distribution values upon several intervals. It allows to understand how memory distribution is influenced by changing the variable on the table axes or diagonals.

Level 3 aims at pattern identification for a given variable. It is an example of *Inter-Experiment* comparison which enables a new kind of global analysis, because it requires to state observation upon the entire set of experiment.

(a) Pattern Recognition Example: Memory Time Domain

Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

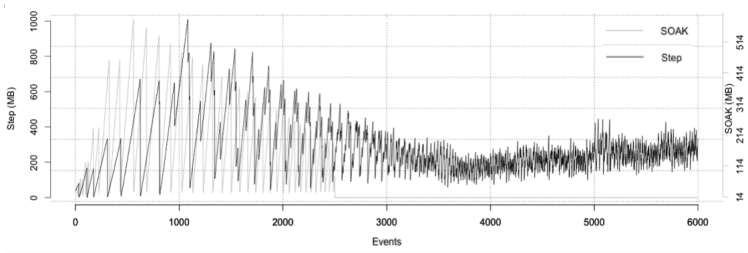
(b) Pattern Recognition Example: Memory Distribution

Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

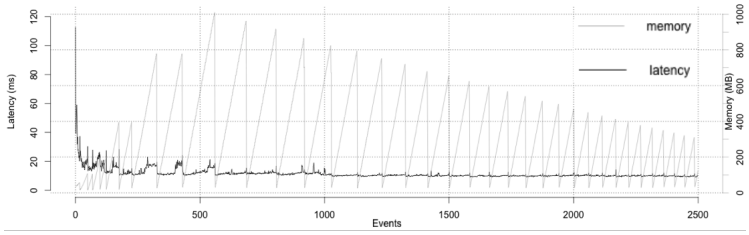
Table 5.2 – Two examples of pattern recognition. Level 2 exploits easy-to-read layouts which highlights the experiment difference to enable *Inter Experiment* comparisons.

Level 3 - Visual Comparison

Finally, Level 3 focuses on single graphical visualisation. Figure 5.15 contains two examples of the possible analysis. Figure 5.15.a shows a case of *Inter Experiment* comparison, highlighting the relation between the same variable over multiple experiments; Figure 5.15.b provides an example of *Intra Experiment* comparison, highlighting the relation between memory and latency within the same experiment.



(a) Multi-Experiment Comparison



(b) Multi-Variables Comparison

Figure 5.15 – Analyser Investigation Stack - Level 3 - Figure (a) shows a case of *Inter Experiment* visual comparison of the memory usage while Figure (b) presents a case of *Intra Experiment* comparison of latency and memory.

Chapter 6

Evaluation

In this chapter, we present an evaluation of $\mathcal{H}$eaven Baselines (See Chapter 4.2) in order to demonstrate how $\mathcal{H}$eaven can extend the traditional hypothesis-based research, towards the Systematic Comparative Research Approach that we presented in Chapter 3. The content of this chapter is organised as follows:

In section 6.1, we describe the method we use to design experiments for RSP Engine testing, and two test sets: SOAK Tests and Step Response Stress Tests. A complete description of the assumptions behind these two experimental sets can be found in Section 6.2.1 for the SOAK Test set and in Section 6.2.2 for the Step Response one.

With the SOAK Test set, we aim at providing a simple use case with naive hypothesis, which highlights the limitations of the traditional top-down analysis approach applied to the SR research and which, consequently, demonstrates the fundamental role of $\mathcal{H}$eaven for RSP Engines evaluation. The Step-Response tests are designed with the goal to show $\mathcal{H}$eaven potential and to provide an alternative use case, which shows $\mathcal{H}$eaven flexibility in supporting the Systematic Comparative Research Approach.

Finally, we present the experiment results respectively for SOAK test, in Section 6.3, and for the Step Response Test, in Section 6.4.

6.1 Experiment Design

Experiment Design (ED) means defining our experiment in order to prove or refute one or more hypothesis, evidencing the behaviour of the system in a controlled execution environment. ED starts with some assumptions, which

compose the Experimental Setting. Researchers formulate hypothesis on the base of these assumptions, Figure 6.1 summarises the collocation of ED within the Top-down approach.

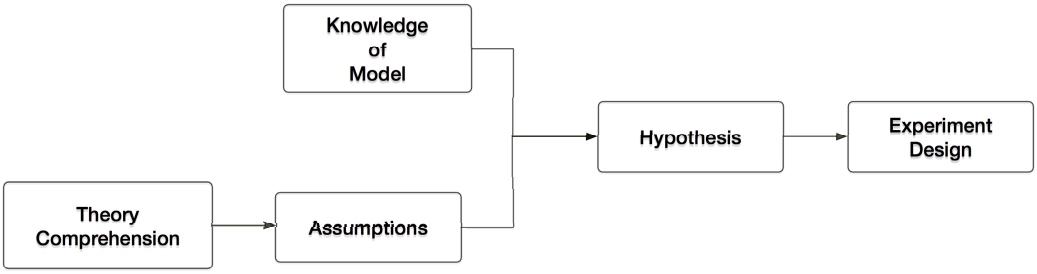


Figure 6.1 – The traditional top-down research designs experiments in order to confirm or refuse hypothesis. The process starts from the theoretical knowledge, through which it is possible to formulate assumptions. Hypothesis are formulated observing an exiting model and considering the assumptions as known truths that simplify the environment. The design of a certain experiment depends on variations of the assumptions or on the variables involved.

From a theoretical point of view we decide to study RSP Engine facing their nature of dynamic systems. (see Section 2.3). Experiment design requires to point out which variable are observed. We simplify the study analysing their behaviour in term of Latency and Memory, which is, together with results completeness and soundness, the minimal meaningful measurement set for an RSP Engine (Chapter 4).

In Chapter 4, we describe the experiment tuple: $< \mathcal{D}, \mathcal{T}, \mathcal{E}, \mathcal{Q} >$ as:

- $\mathcal{E}$ is the RSP Engine,
- $\mathcal{D}$ is the Dataset,
- $\mathcal{T}$ is the Ontology and
- $\mathcal{Q}$ is the Query that $\mathcal{E}$ continuously answers.

In this section, we present our choices about the four elements that compose an experiment, upon which the evaluation is built.

6.1.1 Engine $\mathcal{E}$

The architecture complexity of mature RSP Engines like the C-SPARQL Engine or CQELS is high, and it cannot be easily faced in order formulate hypothesis of comparison. However, the purpose of this evaluation is showing how $\mathcal{H}$eaven can improve the research of RSP Engine. Thus, we can simplify the research survey evaluating less complex systems. To this extent we design and implement the Baselines (Section 4.2) and we evaluate them as the RSP Engines $\mathcal{E}$ subject of our experiments.

In Chapter 3, we describe which requirements guarantee that an RSP Engine is a baselines: Simplicity, Elementarily, Relevance and Eligibility (SERE properties) legitimate the evaluation. Thus, the Baselines can be considered as a simple term of comparison for further research on Stream Reasoning systems and this investigation can be followed as a guideline.

The four baselines differ for two characteristics: RDF Stream Model and Reasoning architecture. Table 6.1 summarises these few but well determined differences, naming the four baselines for the evaluation:

	Naive	Incremental
Graph	GN	GI
Triple	TN	TI

Table 6.1 – We name each baselines according to the reasoning approach and the RDF Stream encoding. Thus an Incremental approach with a graph-based RDF Stream becomes GI.

Among these four configurations we can formulate simple hypothesis, stating which approach is better than an other one within an experiment. Notice that we have a complete model of the baseline systems and we also know many implementation details that can help during the analysis. We can exploit the know-how about their internal mechanisms, described in Section 5.3, to lead our analysis from hypothesis formulation towards empirical results.

6.1.2 Dataset $\mathcal{D}$ and Ontology $\mathcal{T}$

The RDF Streams (see in Section 2.3.1) $\mathcal{D}$ used in the experiments are obtained streaming in different ways the data generated with LUBM (see Section 2.6.2).

Consequently, we chose LUBM ontology[1] as $\mathcal{T}$ for all the experiment. We assume that the ontology does not change over time, therefore the materialisation of $\mathcal{T}$ is computed before starting the experiment and the RSP engine does not have to perform this task.

It is worth to discuss the choice of using data from LUBM rather than SRbench or LSbench. The first one has data, which are not adequate for the experiments, since they do not require any reasoning. The SRbench data, on the contrary, requires reasoning, but, being real-data, do not have the possibility to be scaled up and down nor the flow rate can be adapted. Further details on SRbench and LSBench can be found in Section 2.6.4. Moreover, LUBM has been already used in the Stream Reasoning context [48].

Being LUBM static data, we exploit the *RDF2RDFStream* component of the Test Stand that takes care to adapt the data generate by LUBM to a streaming scenario (see Section 5.2.1). *RDF2RDFStream* is responsible to build $\mathcal{D}$ w.r.t. $\mathcal{T}$, scaling both in terms of dimension of the dataset and the reasoning effort. The component can be set up to obtain an RDF Stream where the number of triples with the same timestamps follows a given distribution.

6.1.3 Query $\mathcal{Q}$

The query $\mathcal{Q}$ used in our experiment depends on the reasoning approach of the RSP Engine in use. Actually, all experiments use variants of the same basic identity query that continuously asks for the materialisation of the active sliding window ω. These variants differ for the sliding factor β.

REGISTER QUERY Q AS
SELECT ?s ?p ?o
FROM STREAM S [RANGE ω STEP β]
WHERE ?s ?p ?o
UNDER ρDF ENTAILMENT REGIME

Query: 6.1.3: Query $\mathcal{Q}$ registered to the $\mathcal{H}$eaven Baselines.

[1] http://swat.cse.lehigh.edu/onto/univ-bench.owl

The entailment regime of the RSP Engines influence performances. For this reason we take for the Baselines ρDF, an RDF-S fragment that reduces complexity while preserving the normative semantics and core functionalities [39]. Several works in the field [48, 35] choose ρDF, because it is the minimal meaningful task for a Stream Reasoner.

6.2 Experiment Set-Up

The *RDF2RDFStream* allows to control the triple distribution in the RDF Stream, thus it is possible to build experiment upon this distribution.

We design set of SOAK Tests to evidence Baselines dynamics and, thus, evaluating their performances.

In order to show $\mathcal{H}$eaven potential, we design a small set of Stress Test, which belongs to the Step Response subcategory. In summary:

- **SOAK**: the number of contemporary triples in the RDF Stream does not change during the experiment.
- **Step Response** the number of contemporary triples in the RDF Stream does not changes for a certain period of time, which guarantees the system to reach the Steady State condition, then it suddenly increase or decrease. The number of triples in the RDF Stream does not changes any more, giving to the system the possibility to reach the Steady State condition again.

The queries registered to $\mathcal{H}$eaven Baselines during all the experiments variate for the size of the sliding window ω. In particular, we use windows in which ω is an integer multiple of the slide parameter β of the window, i.e., it holds that $\omega = \beta * N$. In other words, N is the number of CTEvents that the window contains.

Section 5.3 shows how the proposed baselines take advantage of the ability of Esper to be temporally controlled by an external agent[2] which sends timekeeping events to synchronise the internal time flow. All the triples in the

[2] `http://esper.sourceforge.net/esper-0.7.5/doc/reference/en/html_single/index.html#api-controlling-time`

CTEvent are consider contemporary by the baselines and each *CTEvent* can be seen as a proxy for the timing event.

External time keeping and the *RDF2RDFStream* makes it possible to estimate the content of the active window in terms of number of RDF Triples in any moment of the experiment.

All experiment are execute 10 times to reduce the presence of the outlier.

The following sections contain further details about the two test sets, respectively in Section 6.2.1 for SOAK Test and in Section 6.2.2 for Step Response tests.

6.2.1 SOAK: Tests and Hypothesis

SOAK testing shows the system dynamics, injecting into the RSP Engine in use a constant and continuous input flow (see Section 2.5). We maintain constant the number of contemporary triples in the RDF Stream through the *RDF2RDFStream* module.

SOAK Experiment are 30000 CTEvents long. Each event contains a fixed number of RDF triples, which depends on the specific experiment. Unlikely, it is not possible to foretell how many events are required to reach the Steady State condition for each variable involved, especially memory. Multiple attempts and empirical evaluations are the only way to set up the correct duration.

CTEvent	Number of Slot				
Size	1	10	100	1000	10000
1	1	10	100	1000	10000
10	10	100	1000	10000	
100	100	1000	10000		
1000	1000	10000			
10000	10000				

Table 6.2 – The number of triples in the window in the fifteen SOAK tests as a function of the window size (in terms of N) and of the triples in each CTEvent.

Table 6.2 presents the fifteen SOAK tests we run for. The columns of the table are the different window sizes measured in terms of the values assumed by

Triple in	Number of Slots				
Window	1	10	100	1000	10000
1	1				
10	10	1			
100	100	10	1		
1000	1000	100	10	1	
10000	10000	1000	100	10	1

Table 6.3 – The number of triples in a CTEvent in the fifteen SOAK tests as a function of the window size (in terms of N) and of the total triples in the active window, assuming one and only one CTEvent per slot.

Triple in	Number of Slots				
Window	1	10	100	1000	10000
1	1				
10	2	6			
100	3	7	10		
1000	4	8	11	13	
10000	5	9	12	14	15

Table 6.4 – The enumeration of the fifteen SOAK tests with the layout out Table 6.3.

the number of slots N (see Section 6.1.3). Being $\beta = 100$ ms., they correspond to a window that spans 100 ms., 1 sec., 10 sec. and 100 sec.. The rows are the different number of triples in each *CTEvent* sent by the RDF2RDFStream to $\mathcal{E}$.

Table 6.3 is an alternative layout where the columns of the table still contain the different window sizes measured in terms of the values assumed by the number of slots N, while the rows are the different number of triples in the active window. This layout allows to highlight the window dimension, focusing on the element which actually influences the reasoning.

Table 6.4 shows the enumeration of the fifteen SOAK test exploiting the layout of Table 6.3.

Following the traditional research method, we formulate two naive hypothesis based on the knowledge of the easiest RSP Engine model (see in Section 4.2).

From theory we know that the incremental maintenance of the materialisation is more convenient when the dimension of the changes is small w.r.t the entire window, otherwise for large changes it is more computationally expensive than naive materialisation [18, 43, 48]. This knowledge allow us to formulate a first hypothesis, which can state that the incremental reasoning approach is always better than the naive one, notably for small changes we consider small changes equal or less 20% of the active window content.

From recent works like [8] we know that the graph data structure may speed up reasoning when it contains multiple triples, but it does so introducing an overhead that may hinder performances when it contains few triples. Thus, we can formulate a second hypothesis, which affirms that the triple-based model for RDF Stream is always better than the graph-based one when the number of triples in the graphs is small, as in the previous hypothesis we consider 20% of the active window.

More formally the hypothesis to verify with SOAK experiments are:

- **HP.1** The Incremental reasoning approach is always better than the naive one for small changes.
- **HP.2** The Triple-based model for RDFStream is always better than the Graph-based one if we consider few triples, to compose each graph.

6.2.2 Step Response Tests

Step Response testing allows to see how the system reacts to a sudden changing in the input condition (see Section 2.5). They are related to SOAK test, because they allow to study deeply the initial warm-up phase of the system. This set of experiments evidences the different responses of an RSP Engine if we move from an input condition to another instead of starting from scratch.

Step Response experiments are 40000 CTEvents long. Since this duration requires hours of execution we investigate only few configurations, all with a slots number of 10. Table 6.5 summarises the step experiment set-up, where the step is positioned at the half of the execution, 20000 events.

We do not formulate hypothesis for Step experiments. The purpose of this test set is to show $\mathcal{H}$eaven potential in term of analysis an experiment design.

	CTEvent size	
Number of Slots	Initial Size	Final Size
10	10	100
10	100	1000
10	10	1000

Table 6.5 – Step Response Test Summary Table - the first column indicates that every Step Response test is executed with a number of slot equals to 10. At the beginning of the experiment the CTEvent sizes are indicated in the second column, while their sizes at the end of the experiment is reported in the third and last column. All the Step Response tests are 40.000 CTEvents long, and the step position is always at the half of the experiment duration, thus 20.000 CTEvents

They shows how it is possible to set up $\mathcal{H}$eaven in order to stress an RSP Engine in different ways.

6.2.3 Execution Environment

Before presenting experiment results, a brief description of the execution environment is required. All experiment are executed exploiting a dedicated machine, an iMac mid-2011 with 12GB RAM and 3.6 Ghz of a Intel i5 64 Bit, which run OS X 10.10.2 Yosemite. Since $\mathcal{H}$eaven is developed with Java 7[3], we use the version 1.7.0.71 of the JVM. The execution happens in a controlled environment, which tries to reduce the number of disturbing elements like network, graphical interface and other running processes.

6.3 SOAK Test Evaluation Results

In this section, we analyse the results of the SOAK Test experiments, exploiting the investigation stack presented in Chapter 4. The analysis goes through the four levels of the stack, with the aim of confirming or refusing the hypothesis presented in Section 6.2.1.

The content of this section is organised as follow: Section 6.3.1 presents the output of the *Steady State Identification* block. Section 6.3.2 presents

[3]http://www.oracle.com/technetwork/java/javase/javase7locales-334809.html

the results obtained at Level 0, the dashboard visualisation. Section 6.3.3 provides a qualitative comparison for latency and memory, built at Level 1. Section 6.3.4 contains example of pattern identification at Level 2. Finally, in Section 6.3.5, we include some examples of the Level 3 *Intra Experiment* comparisons and in Section 6.4 we include examples of *Inter Experiment* ones w.r.t the Step Response tests.

6.3.1 Steady State Identification Block Results

The *Steady State Identification* (SSI) block, as reported in Chapter 5, annotates the averaged raw data received by the *Pre-Processing* block, indicating which variable has reached a Steady State condition for an experiment. Table 6.6 contains all the results the SSI Block, distinguishing between latency and memory for each variables.

			Steady State Condition Identification Result							
	CTEvent		GN —		GI —		TN —		TI	
EN	Size	Slot Num.	Lat	Mem	Lat	Mem	Lat	Mem	Lat	Mem
1	1	1	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
2	10	1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
3	100	1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4	1000	1	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
5	10000	1	NA	NA	NA	NA	NA	NA	NA	NA
6	1	10	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
7	10	10	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
8	100	10	Yes	No	Yes	No	Yes	No	Yes	No
9	1000	10	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
10	1	100	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
11	10	100	Yes	No	Yes	No	Yes	No	Yes	No
12	100	100	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
13	1	1000	Yes	No	Yes	No	Yes	No	Yes	No
14	10	1000	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
15	1	10000	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 6.6 – Steady State Identification Block - Output Summary Table - Experiment number follows the layout of Table 6.3, filling the columns from the top to the bottom. The Steady State condition is evaluated after the average value of 25.000 CTEvents, corresponding to the 85% of the experiment duration. It is an enough for the Steady State evaluation of latency, even in case of filling phenomena, but it is not for memory in several experiments.

All the SOAK Experiments reach the Steady State condition for latency, but some of them did not for memory. Notice in particular that GI does not

reach the memory Steady State for experiment 1 (CTEvent Size = 1000 and a number of slot = 1). Moreover, all the baselines do not reach the memory Steady State for experiments 8, 11, 13 (CTEvent Size = 1000 and a number of slot from 10 to 1000).

Experiment number 5 (the CTEvent Size = 10000 a slot Nnumber = 1=) did not terminate correctly for all the baselines, due to an execution error and thus, its results are not part of our evaluation.

6.3.2 Level 0 - Dashboard Views

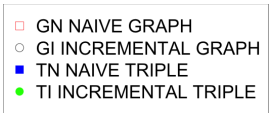


Figure 6.2 – Dashboard Legend

The SOAK test result analysis starts with the dashboard view at Level 0. Each dashboard presented is composed by two charts which show, in different form, the relation between the different four baselines. Figure 6.2 shows the dashboard legend, which is fixed for all the charts related to the SOAK tests.

Dashboard One - Fixed Size CTEvent = 1

Figures 6.4 shows how the behaviour of the four Baselines change between a set of experiments, which have a fixed CTEvent size of 1 triple and variates the number of slot in the active window, from 1 (Tumbling) to 10000. This configuration means moving on the latest diagonal of Table 6.3.

In Figure 6.4, we can appreciate that increasing the number of slots in the active window, and thus the window size in terms of triple, all the baselines show latency worsening. The Baselines which exploit the Incremental reasoning approach, GI and TI, are not represented in the figure when the number of slides is equal to one (Tumbling window of only one triple), because their latency it too small to be represented in this scale. Increasing the number of slot, GI and TI have the bigger worsening in term of latency than GN and TN. Actually, they do not became worse, but the Incremental approach is at least comparable with the naive one, when the number of slots became high (10000) and the variation is about 2/10000.

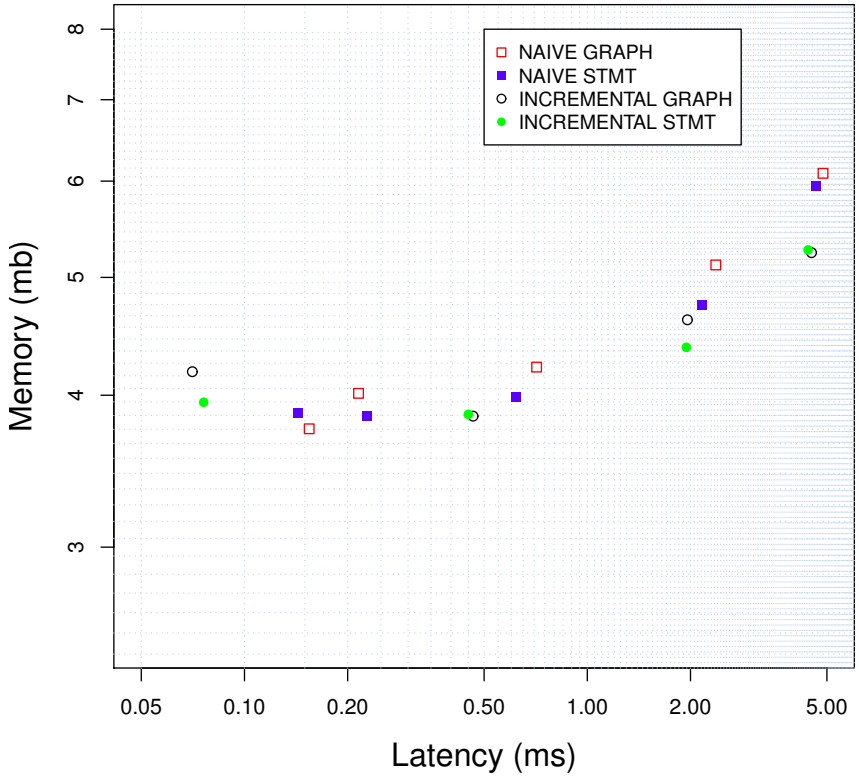


Figure 6.3 – Analyser Investigation Stack - Level 0 - Dashboard One - The figure shows how the Baseline performances (avg memory (y) and latency (x)) variate between a subset of SOAK Test experiments, composed by experiments with a constant CTEvent Size (K) = 1 triple (Experiment (EN) 1,6,10,13,15).

Observing Figure 6.3, which shows all the experiment within a single image, we can also individuate the worsening for the memory consumption. A first obvious insight may be that *Bigger problem requires in general more resources.*

Dashboard One (Figure 6.4 and Figure 6.4) confirms Hypothesis [Hp.1], the incremental approach performs better than the naive one for both latency and memory, and it partially confirms [Hp.2]: the latest results in Figure 6.4 shows that the memory usage depends on the reasoning approach and thus the relation between graph-based streams and triple-based one depends on it too. The figure confirms [Hp.2], the latency is smaller in experiments with a big window and a triple-based stream, while for smaller windows a graph-based one may work better.

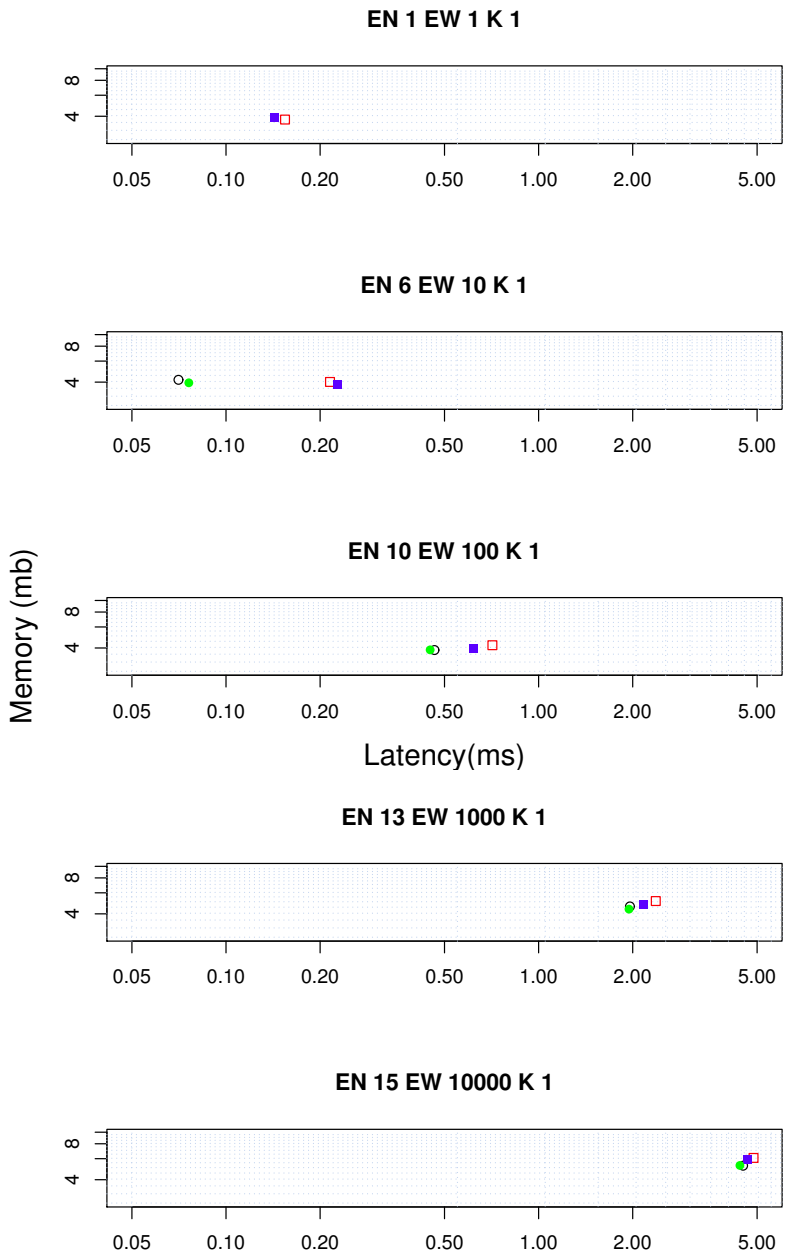


Figure 6.4 – Analyser Investigation Stack - Level 0 - Dashboard One - The figure shows how the Baseline performances (average memory (y) and latency (x)) variate between a subset of SOAK Test experiments, composed by experiments with a constant CTEvent Size (K) = 1 triple (Experiment (EN) 1,6,10,13,15); EW indicates the number of slots in the window. The results are reported comparing experiments on different stand alone plots.

Dashboard Two - Fixed Number of Slot = 10

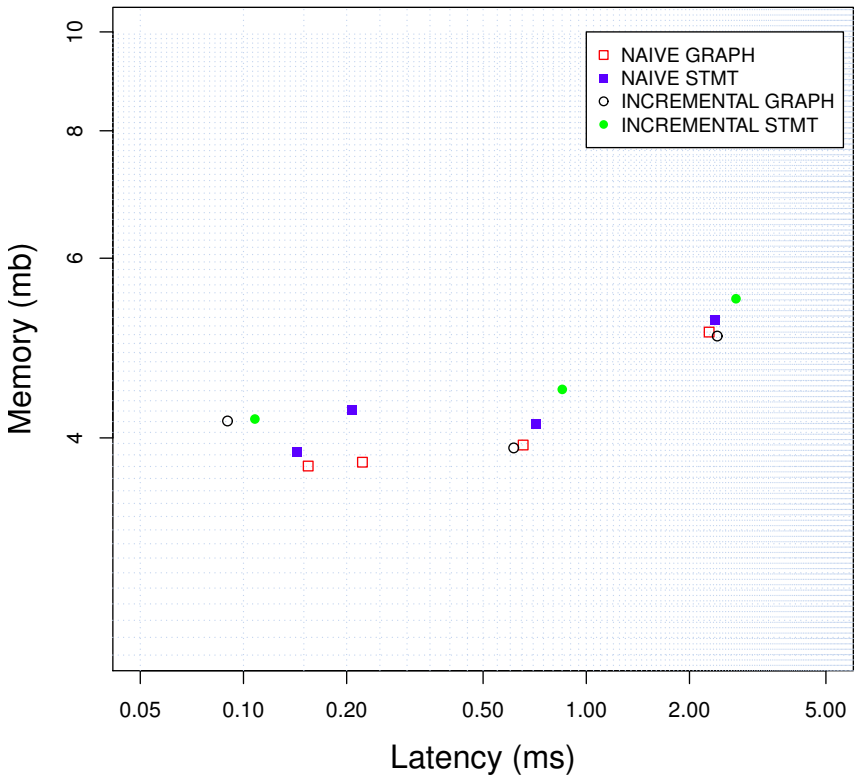


Figure 6.5 – Analyser Investigation Stack - Level 0 - Dashboard Two - The figure shows how the Baseline performances (average memory (y) and latency (x)) variate between a subset of SOAK Test experiments, composed by experiments with a constant number of slots in window (EW) of 10 (Experiment (EN) 6,2,8,9).

Figure 6.6 shows how the behaviour of the four Baselines changes between a set of experiments which have a fixed number of 10 slots in the active window, while the size of the CTEvent changes from 1 to 10000. This analysis means, w.r.t Table 6.3 layout, moving from the top to the bottom of the second column.

From the observation of the two figures emerges that: (1) Latency worsening is still clearly visible in Figure 6.6, while the memory ones requires Figure 6.5; (2) The behaviour of the baselines becomes indistinguishable when the window size in terms of triple become 10000.

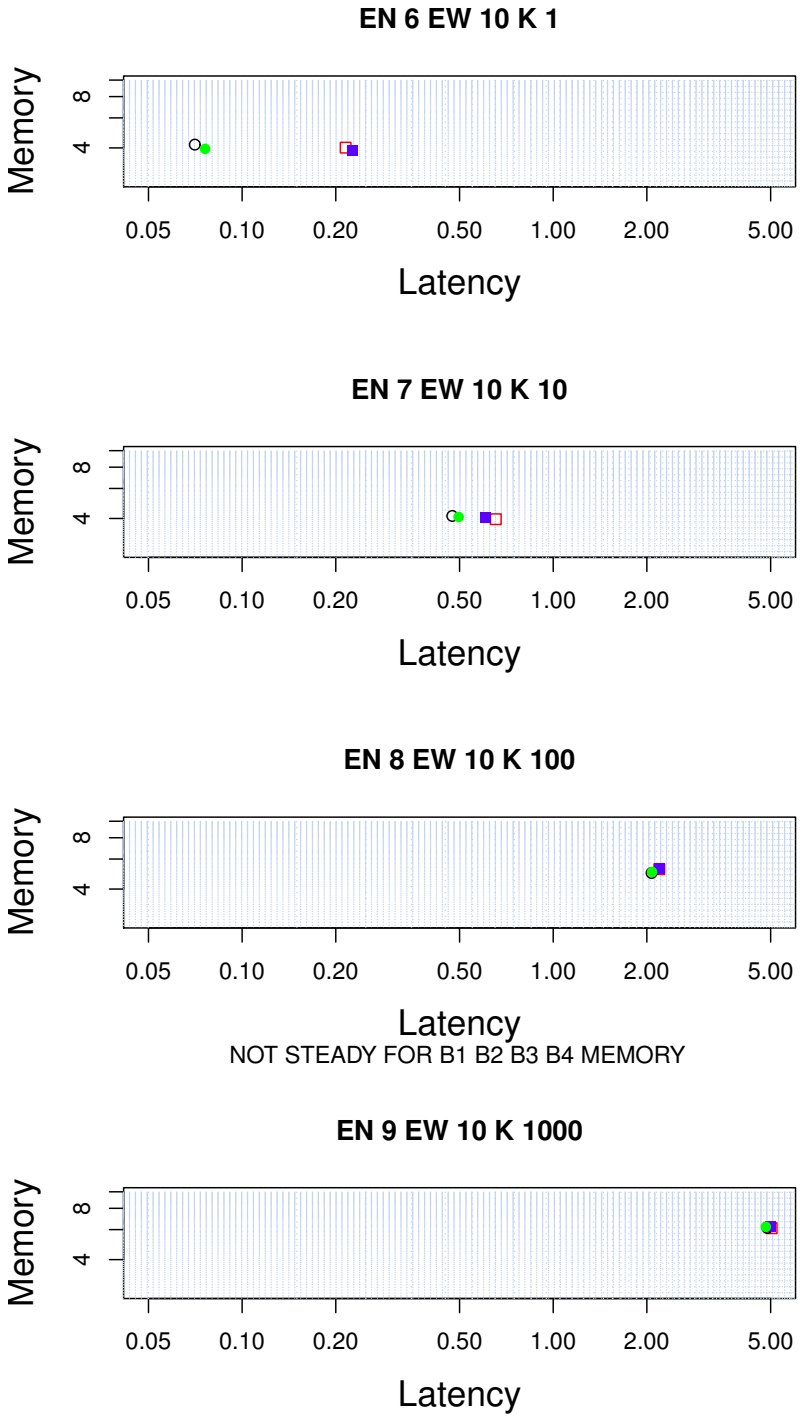


Figure 6.6 – ANALYSER Investigation Stack - Level 0 - Dashboard Two - The figure shows how the Baseline performances (average memory (y) and latency (x)) variate between a subset of SOAK Test experiments, composed by experiments with a constant number of slot in window (EW) of 10 (Experiment (EN) 6,2,8,9). K indicates the CTEVENT size. The results are reported comparing experiments on different stand alone plots.

[Hp.1] is confirmed for the latency performance, as can be seen in Figure 6.5, GN and TN are always slower than GI and TI.

For memory, the results are ambiguous and requires further investigations, only 50% of the experiments confirm the hypothesis. Figure 6.5 shows that [Hp.2] is confirmed within a single reasoning approach, while in general the results are similar to the ones of [Hp.1].

Dashboard Three - Fixed Windows Size (Triples) = 10000

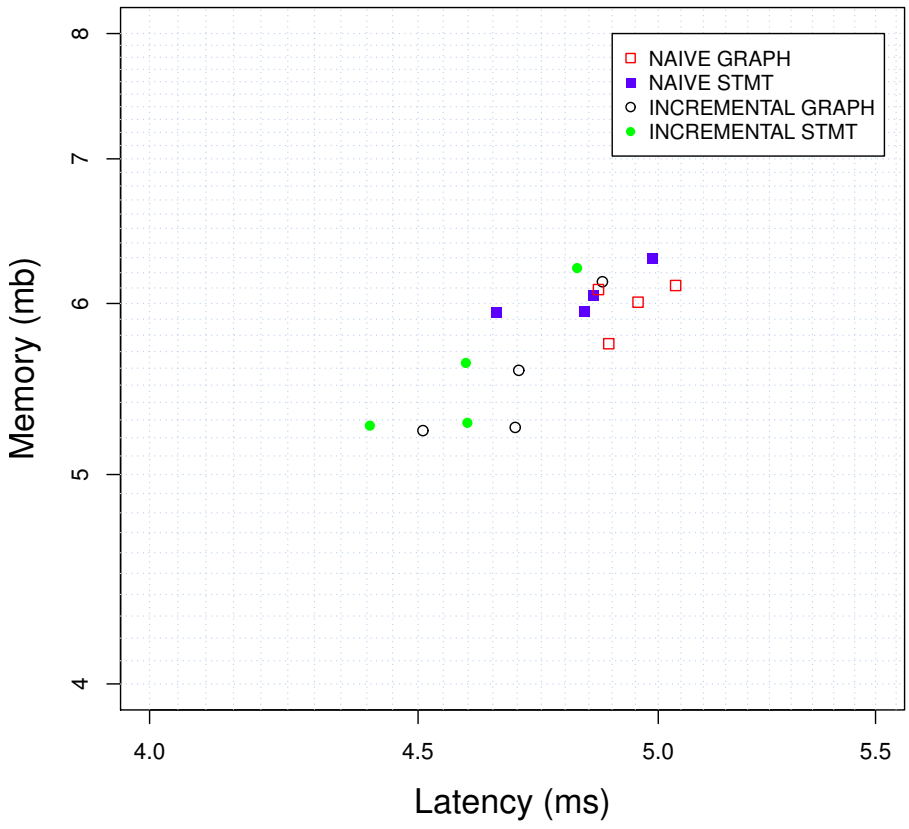


Figure 6.7 – Analyser Investigation Stack - Level 0 - Dashboard Three - The figure shows how the Baseline performances (average memory (y) and latency (x)) variate between a subset of SOAK Test experiments, composed by experiments with a constant number of triple in the active window, thus a constant product of CTEvent Size (K) * Num.Slots (EW) = 10000 (Experiment (EN) 9,12,14,15, experiment 5 is excluded for an erroneous execution).

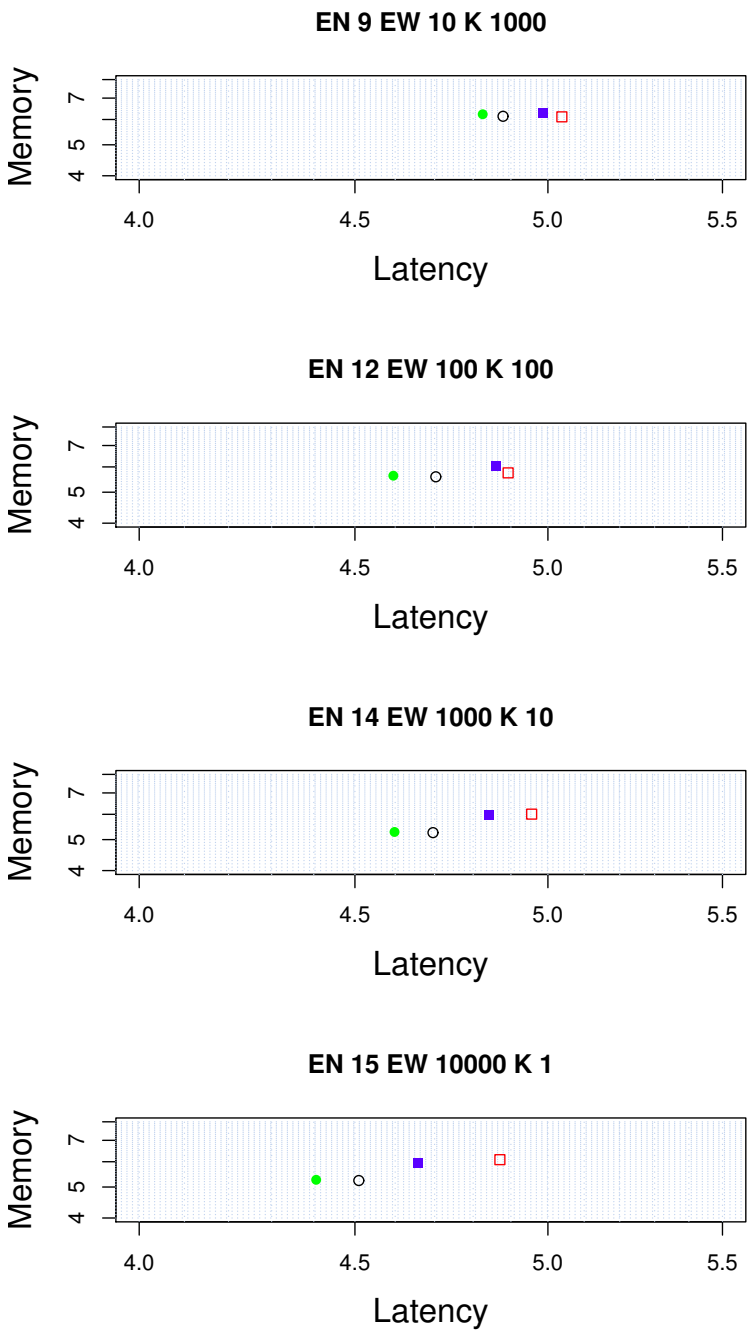


Figure 6.8 – Analyser Investigation Stack - Level 0 - Dashboard Three - The figure shows how the Baseline performances (average memory (y) and latency (x)) variate between a subset of SOAK Test experiments, composed by experiments with a constant product of CTEvent Size (K) * Num.Slots (EW) = 10000 (Experiment (EN) 9,12,14,15 - 5 is excluded for an erroneous execution). The results are reported comparing experiments on different stand alone plots.

Figure 6.8 shows how the behaviour of the four Baselines changes between a set of experiments, which have a constant number triples within the active window equal to 10000, stated by maintaining the product CTEvent size (K) and Slot Number (EW) constant to 10000 triples. Fixing the window size in terms of triple to 10000 means moving from the left to the right on the lowest row of Table 6.3.

The main observation, we can state on Figures 6.8 and 6.7, is that the Baseline performances seem depending only on the dimension of the active windows, indeed for all the observed experiment Baseline memory usage and latency are disposed in a small interval.

[Hp.1] is confirmed for both the memory and the latency as the figures show. However the difference are very small between the approaches. [Hp.2] is confirmed again observing a single reasoning approach at time, while in general it is not possible to say that Triple-based RDF Stream performs better than the Graph-based one.

Level 0 allows to state that there is no strong dominance of a Baseline w.r.t. the other ones in the provided analysis. Moreover, a weak dominance can be observed in different steps both in term for latency and memory. The most clear examples are the medium size problems in Figures 6.4 and 6.3 and in Figures 6.6 and 6.5. However, result hierarchy is not absolute and it may change by changing experiment configurations. This observation refutes explicitly [Hp.2] formulated in Section 6.2.1, while [Hp.1] was not completely confirmed. There are experiments where the naive reasoning approach seems performing better than the incremental one, at least in term of latency.

6.3.3 Level 1 - Statistical Values Comparison

As Expected, the research over RSP Engines requires further analysis to motivate these unexpected findings. Level 1 exploits statistical investigation through an easy to ready layout, which simplify the inter-experiment comparison (see 4.3 Level 1).

The current evaluation operates over the average values of latency and memory. Tables 6.7 and 6.8 contain in all the experiment results respectively for latency and memory at Steady State. Both the tables lay the results out, according with the layout of Table 6.3. We decide to use the qualitative result

(a) Incremental

Triple in Window	Slots Number 1	10	100	1000	10000
1	G				
10	G	≃			
100	G	≃	≃		
1000	G	≃	≃	≃	
10000	NA	T	S	T	T

(b) Triple

Triple in Window	Slots Number 1	10	100	1000	10000
1	I				
10	I	I			
100	N	I	I		
1000	N	I	I	I	
10000	NA	I	I	I	I

(c) Naive

Triple in Window	Slots Number 1	10	100	1000	10000
1	≃				
10	≃	≃			
100	G	≃	T		
1000	G	≃	T	T	
10000	NA	≃	≃	T	T

(d) Graph

Triple in Window	Slots Number 1	10	100	1000	1000
1	I				
10	I	I			
100	≃	I	I		
1000	N	I	I	I	
10000	NA	I	I	I	I

Table 6.7 – Analyser Investigation Stack - Level 1 - SOAK Test average latency comparison trough a qualitative approach. The following convention indicates the baseline has not reached the Steady State Condition: G, T, N, I. (a), (c) - latency results comparison between Incremental and Naive approaches; (b), (d) - latency results comparison between Graph-based and Triple-based models.

representation, but $\mathcal{H}$eaven allows also more detailed analysis with quantitative comparisons as shows in Section 5.4. To properly read the tables note that they report that a baseline is better than another one when the difference in term of latency or memory is bigger than 5%, otherwise we consider the two terms of comparison as equal and we use the simble ≃. Moreover, we indicate that the better solution has not reached the Steady State Condition with the underlined symbols G, T, N, I.

When N >1, the results in Table 6.7.a and 6.7.c allow to say that using a Triple-base RDF stream is faster than Graph-based one. In particular, for the case N=1000 when the window contains 1000 triples (i.e., each CTEvent contains only one triple), the Naive Triple-based approach is about 10% faster than the Naive Graph-based one while the Incremental Graph-based is even about 20% faster. This findings confirm [Hp.2], while the cases when N=10 the does not confirm the hypothesis because the results can be consider as equal (result differences are smaller than 5%). A possible explanation is that

(a) Incremental

Triple in Window	Slots Number 1	10	100	1000	10000
1	T				
10	G	T			
100	G	T	G		
1000	G	G	G	T	
10000	NA	G	G	G	G

(b) Triple

Triple in Window	Slots Number 1	10	100	1000	10000
1	N				
10	I	N			
100	N	N	I		
1000	N	I	I	I	
10000	NA	I	I	I	I

(c) Naive

Triple in Window	Slots Number 1	10	100	1000	10000
1	G				
10	G	T			
100	G	G	T		
1000	G	G	G	T	
10000	NA	G	G	T	T

(d) Graph

Triple in Window	Slots Number 1	10	100	1000	10000
1	N				
10	N	N			
100	$\simeq$	N	I		
1000	$\cong$	I	I	I	
100000	NA	N	I	I	I

Table 6.8 – Analyser Investigation Stack - Level 1 - SOAK Test average memory comparison trough a qualitative approach.The following convention indicates the baseline has not reached the Steady State Condition: G, T, N, I. (a), (c) - memory results comparison between Incremental and Naive approaches; (b), (d) - memory results comparison between Graph-based and Triple-based models

the dimension of the graph cannot be considered small w.r.t the window when N=10.

When N=1 (i.e., the window contains only one *CTEvent*) instead, the results in Table 6.7.b and Table 6.7.d show that for large events the Naive approach is faster than the Incremental one, as we stated when we formulate [Hp.1]. Instead, when *CTEvent* contains only few triples, the Incremental approach is faster and this is not intuitive, because to formulate [Hp.1] we consider the changes dimension in percentage.

The results in Table 6.7.b and 6.7.d support [Hp.1] by stating that when the number of changing triples in $\Delta + \Delta-$ (Section 4.2) is a small fraction of those in the window an Incremental approach is faster than the Naive one. The exception of case N=1, but it can be seen as a limit case, where the reasoner is asked to deduce all the implicit triples implied by the only explicit triple in the window.

While is possible to state meaningful observation over latency data, the same is not possible for memory ones. $\mathcal{H}$eaven shows that the study of the memory can not be faced with the same methods to study latency (comparison of the average values under a clearly identified the Steady State condition).

Table 6.8 reports the results for the memory usage during the experiments. Table entries do not confirm what we observed in Table 6.7 and sometimes it even refutes our findings. It is clear that memory usage does not follow the same behaviour of latency, even if [Hp.1] and [Hp.2] are not totally refused by the results.

Further statistical analysis are certainly meaningful. Comparing maximum or minimum as we did for the average values may detail much more the system memory usage.

6.3.4 Level 2 - Patter Identification

Statistical analysis are meaningful, but may reduce too much the RSP Engine complexity by focusing on single element of comparison. A more complete analysis is required and $\mathcal{H}$eaven can investigate the behaviour of the system over all the experiment execution.

Level 2 exploits again the layout of Table 6.3, disposing in table cell a graphical representation of a certain variable (Latency, Memory). The graphical representation choice depends on the research necessities: time domain representations and value distribution are the ones we explored during our evaluation, both in log scale or linear scale.

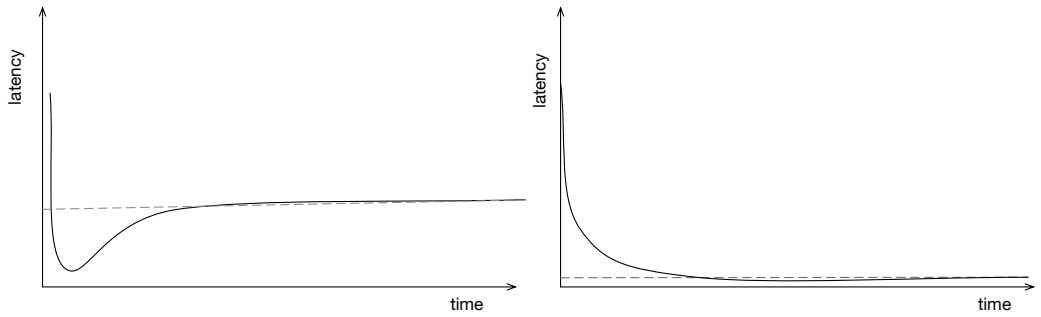


Figure 6.9 – Recognised Latency Pattern for SOAK Experiments

The Tables 6.9 and 6.10 represent in linear scale the latency time series of the four Baselines. Immediately we can see that most of the systems reach a

Steady State condition in a short period, between the 10% and the 20% of the entire experiment duration. Some of them show filling phenomena, when the sizes of the window start to be relevant w.r.t. the experiment duration. In general, the latency trend follows one of the pattern showed in Table 6.9.

The Tables 6.11 and 6.12 show the representation in linear scale of the memory time series of the four Baselines. Despite what happens for latency, we can observe that the memory usage does not reach a Steady State condition for all the experiments (Baseline GI and TI for windows of dimension 1000 Triples with more than one slot). To reach the Steady State condition for memory, the system requires from 3.000 to 25.000 CTEvents, but in general it has not a common behaviour. The variance of these results is high and can be motivated arguing about how Java is managing the memory during the execution. The optimisation policies work better when the system have to handle a lot of triples. The proof of this insight can be seen in Tables 6.11 and 6.12. The lower levels of the Tables reach the Steady State condition before the higher ones. For those experiments that involved GI and TI, the Baselines with an incremental reasoning approach, the Steady State is not even reached, maybe because the memory consume does not alert the JVM at all.

The filling phenomena we identified for latency are still visible in both the tables. This denotes the existence of a relation between memory and latency. This is one of the point we can further investigate through the next analysis step: Level 3.

Tables 6.13 and 6.14 represent the distribution of memory time series values of the four Baselines. To obtain this representation we individuate the minimum and the maximum between all the experiments. We divided the span in some intervals and automatically we count how many values of the memory time series fit the intervals, repeating this count for all the experiment. through this representation we can see how the memory usage is distributed during the experiment execution. We can observe that the memory values for the baselines that exploit incremental reasoning, GI and TI, are distributed in a smaller interval w.r.t the baselines which exploit the Naive reasoning approach and thus GI and T I have a smarter usage of the memory. When the window size increases, the memory consumption shift to the right, towards the intervals with higher values.

All the four tables about memory time series and memory distribution show

that there are strong difference between the experiment subset with window dimension of 1000 triples and the ones with window dimension 10000 triples. This is a meta-insight that improves the Experiment Design model. Actually we are still not able to predict the baselines behaviour, but we can further investigate through $\mathcal{H}$eaven.

The quantitative nature of the hypothesis formulated in Section 6.2.1 does not allow to exploit Level 2 for Hypothesis verification. Actually [Hp.1] and [Hp.2] consider the performance as the main evaluation metric, while Level 2 fulfils the necessity to understand the system nature. The insights above allow to improve the RSP Engine model, upon which is possible to formulate more precise hypothesis and to explain the unpredictable results we obtained from Level 0 and Level 1.

6.3.5 Level 3 - Single Visual Comparison

Level 3 operates a drill down from Level 2, providing examples of *Intra Experiment* comparisons, in order to highlight the relation between memory and latency performances. Level 2 has shown several differences that can explain the absence of a dominant Baseline solution over all the experiment and w.r.t [Hp.1] and [Hp.2]. Thus Level 3 starts from three important findings:

- All the experiments reach the Steady State for latency, but several of them do not reach it for memory.
- When the Steady State condition is reached: latency requires about 2.000-5.000 CTEvents w.r.t the total experiment duration of 30.000, while memory requires from 5.000 to 25.000 CTEvents.
- Both memory and latency analysis show the presence of filling phenomena, which should be further investigated.

Figure 6.12 contains two examples of experiment that reach the Steady State condition for latency but not for memory. Figure 6.12.a shows the latency and memory usage for the Baseline GN within the experiment eleven (CTEvent size = 10, Number of Slot = 100), Figure 6.12.b shows the latency and memory usage of the Baseline TI, within the same experiment.

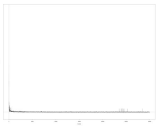
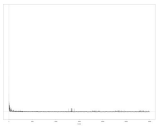
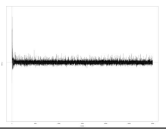
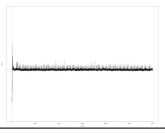
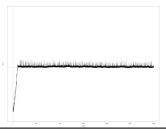
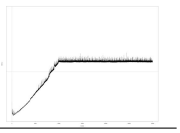
If we consider only the memory trend, we note an oscillating behaviour which contrast with the idea of Steady State. A more careful analysis identifies

the presence of some spikes in the latency time series. It represents a certain period of time where the content of the window requires a strong reasoning effort. This may also influence the memory usage. Moreover, this kind of analysis can not be seen from higher level.

Figure 6.11 (a) and (b) show two examples of steady state condition reaching. The figure 6.11.a describes that the Baseline GN for experiment seven (CTEvent size = 10, Number of Slot = 10) reaches the Steady State around the 10% of the entire experiment duration for latency and about 45% for memory. Java probably focuses on speeding up the execution rather than saving resources. A completely different behaviour can be appreciate in the subfigure (b), where both latency and memory reach the Steady State around the 10% of the entire execution. It seems that bigger is the amount of resources required by the program, faster Java's optimization policies work.

Figures 6.12 (a) and (b) show the relation between the filling phenomena together for latency and memory for the Baselines TN and TI within Experiment 15 (CTEvent size = 1 Num. Slots = 10000). The main insight regards how Java manages the initial warm-up phase. We have seen that the JVM usually over-estimates the necessary amount of memory at the beginning of the execution, then it tries to optimize. When the window dimension in terms of triple is increasing the memory occupation has a worsening and latency follows the same behaviour. Finally, latency and memory reach the stability when the filling phenomenon ends, since the number of triples in the active window is now constant.

(a) Graph Naive

Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

(b) Graph Incremental

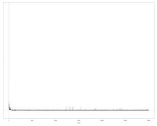
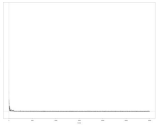
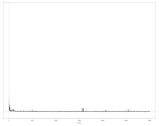
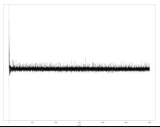
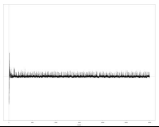
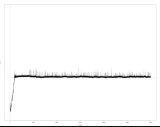
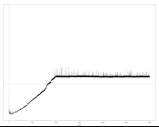
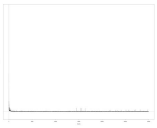
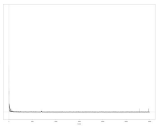
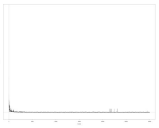
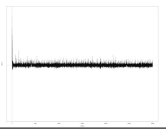
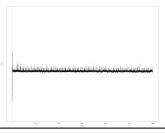
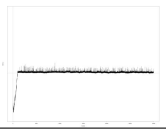
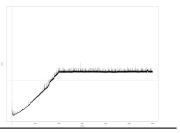
Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

Table 6.9 – The figure shows the representation in the time domain of latency for GN (a) and GI (b).

(a) Triple Naive

Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

(b) Triple Incremental

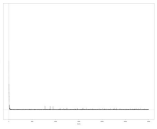
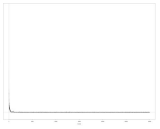
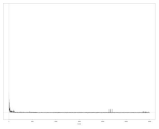
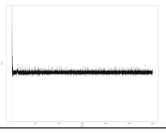
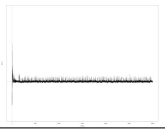
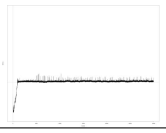
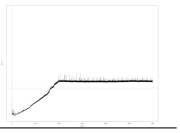
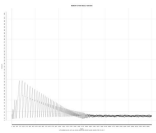
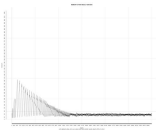
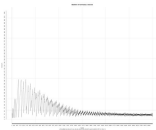
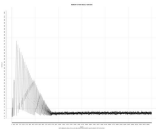
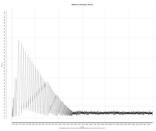
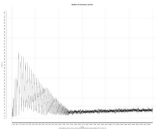
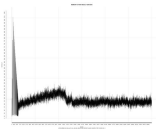
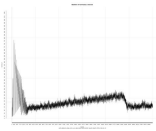
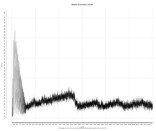
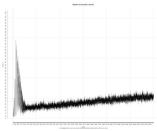
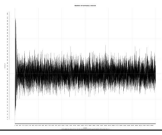
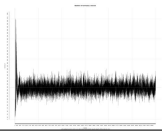
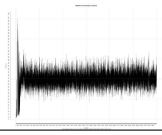
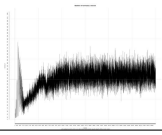
Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

Table 6.10 – The figure shows the representation in the time domain of latency for TN (a) and TI (b).

(a) Graph Naive

Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

(b) Graph Incremental

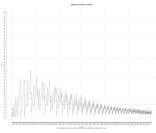
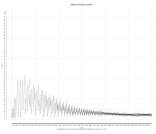
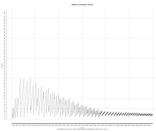
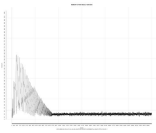
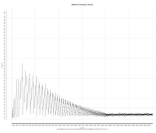
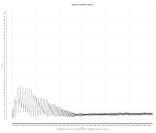
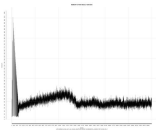
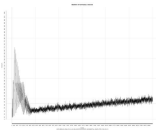
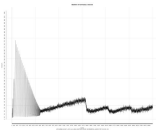
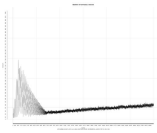
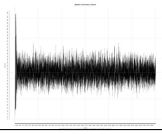
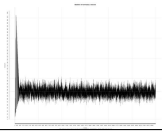
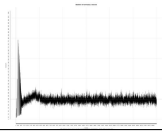
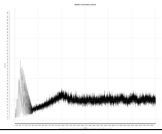
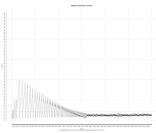
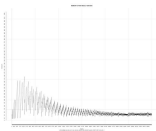
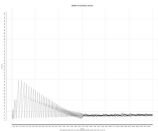
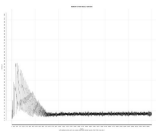
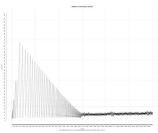
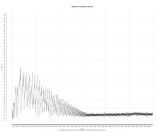
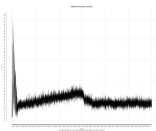
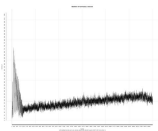
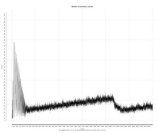
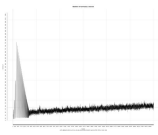
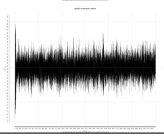
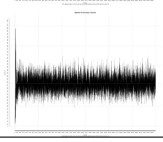
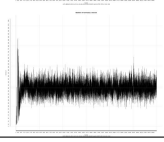
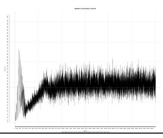
Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

Table 6.11 – The figure shows the representation in the time domain of memory for GN (a) and GI (b).

(a) Triple Naive

Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

(b) Triple Incremental

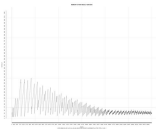
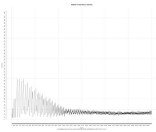
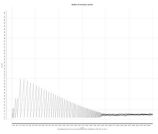
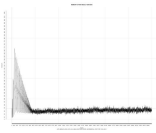
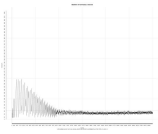
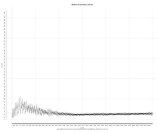
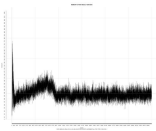
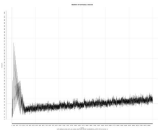
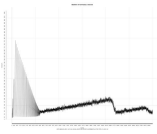
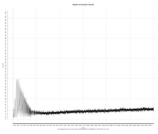
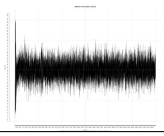
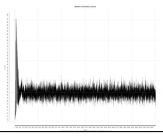
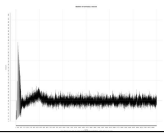
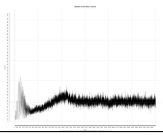
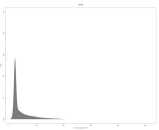
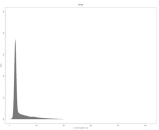
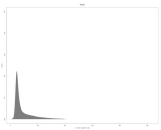
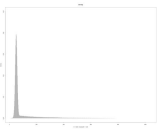
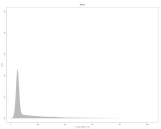
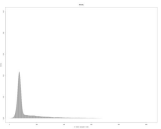
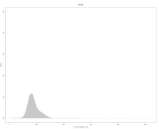
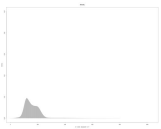
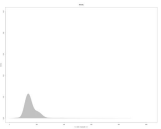
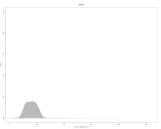
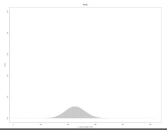
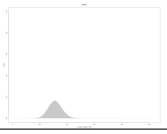
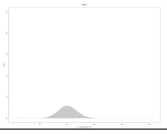
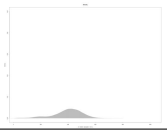
Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

Table 6.12 – The figure shows the representation in the time domain of memory for TN (a) and TI (b).

(a) Graph Naive

Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

(b) Graph Incremental

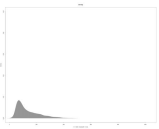
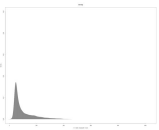
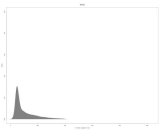
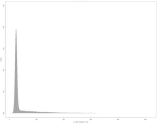
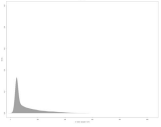
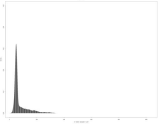
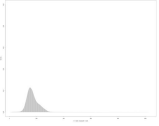
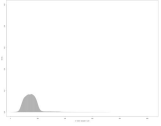
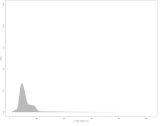
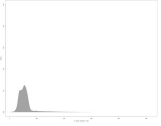
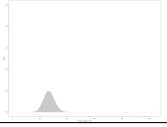
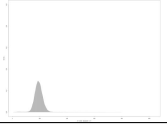
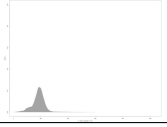
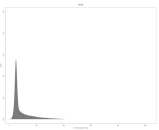
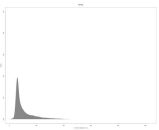
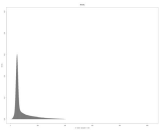
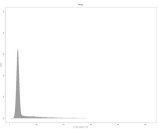
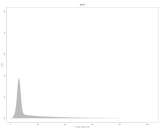
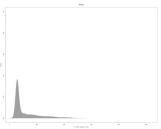
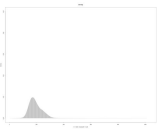
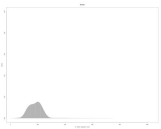
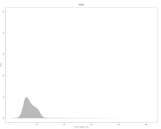
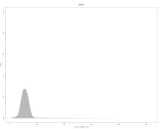
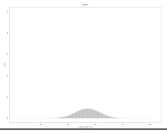
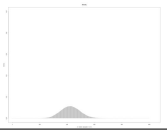
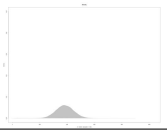
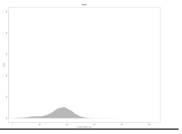
Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

Table 6.13 – Figure shows how memory values for GN (a) and GI (b) are distributed over ten intervals between the global minimum and maximum of all the SOAK tests results.

(a) Triple Naive

Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

(b) Triple Incremental

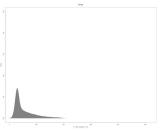
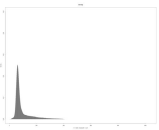
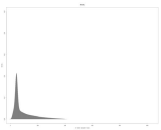
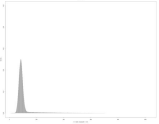
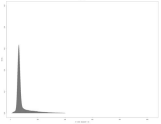
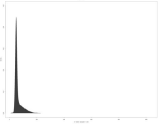
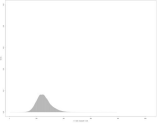
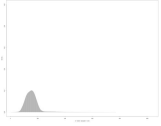
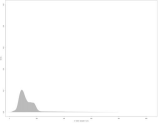
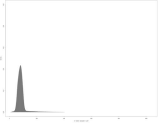
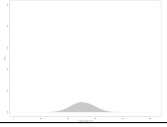
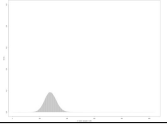
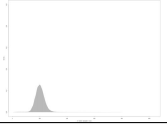
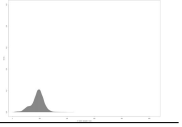
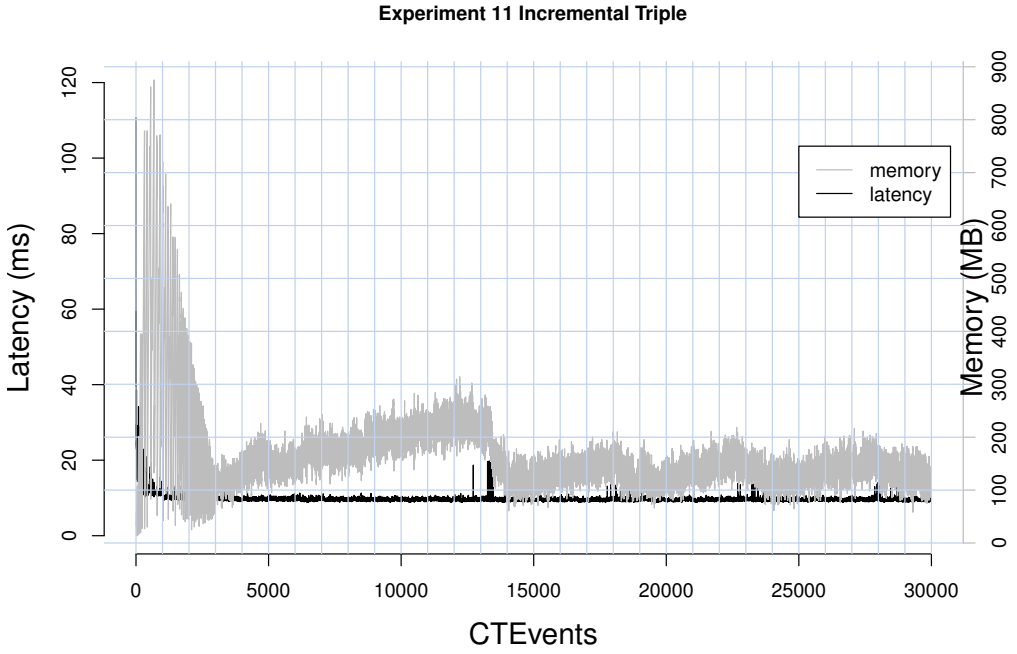
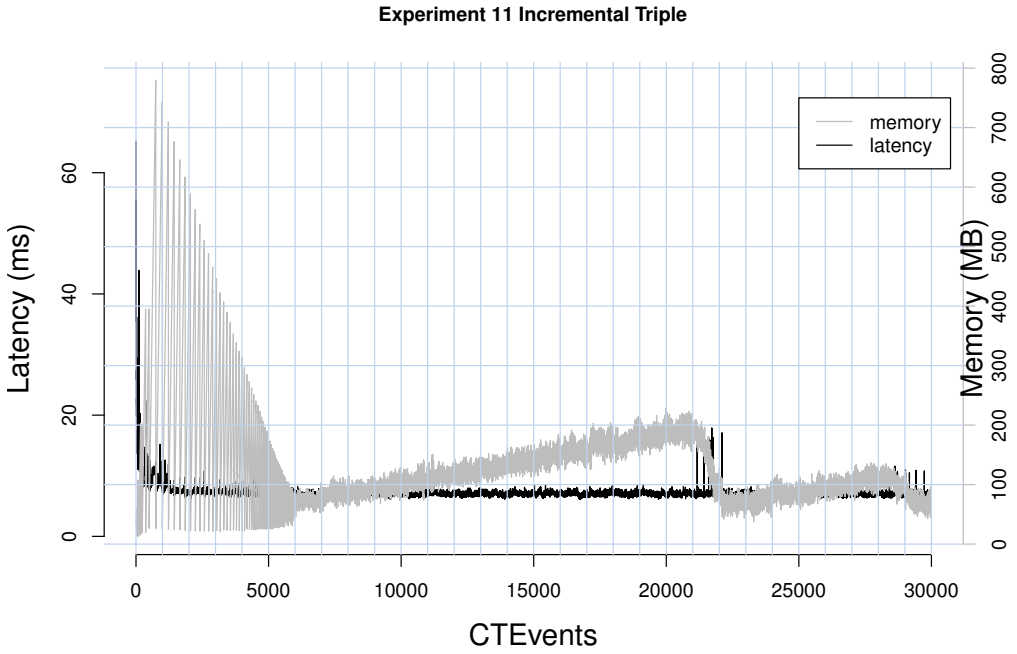
Triple in Window	Slots Number 1	10	100	1000	10000
1					
10					
100					
1000					
10000					

Table 6.14 – Figure shows how memory values for TN (a) and TI (b) are distributed over ten intervals between the global minimum and maximum of all the SOAK tests results.

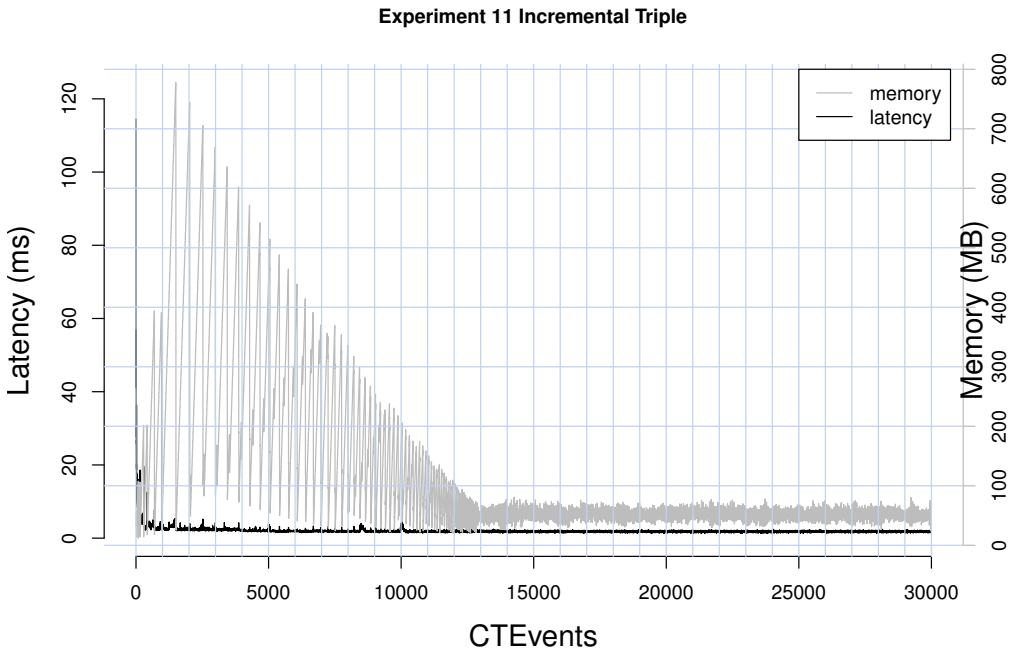


(a) GN Exp 11, CTEvent Size = 10 Num.Slot = 100

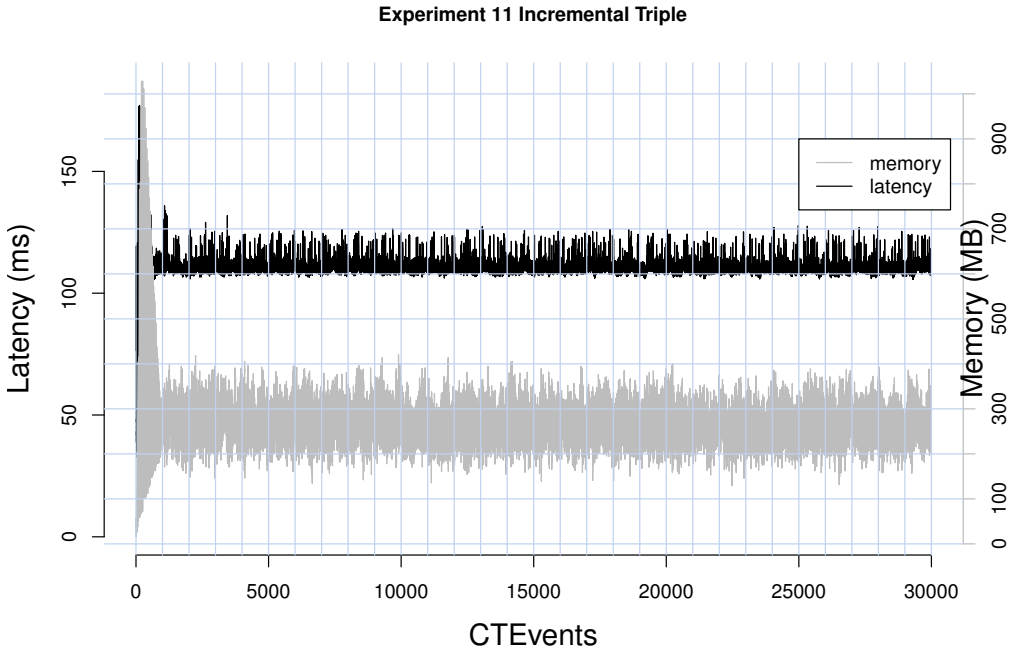


(b) TI Exp 11, CTEvent Size = 10 Num.Slot = 100

Figure 6.10 – Analyser Investigation Stack - Level 3 - Memory (Y) and latency (Y) behaviours in time domain (X) - SOAK Test Exp 11, CTEvent Size = 10 Num.Slot = 100, w.r.t baseline GN (a) and TI (b).

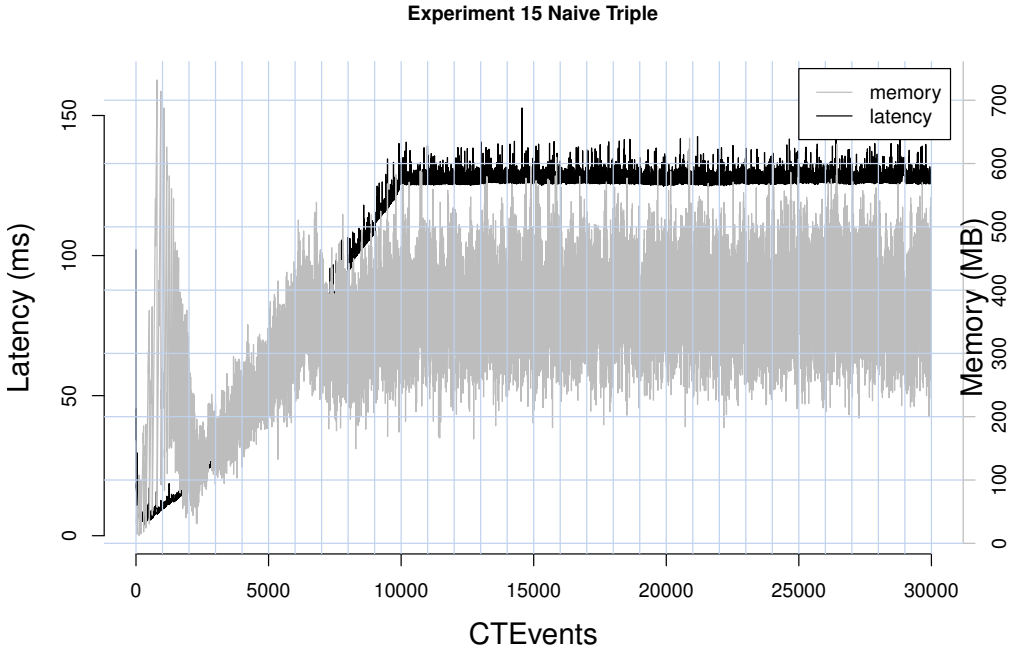


(a) GN Exp 7, CTEvent Size = 10, Num.Slot = 10

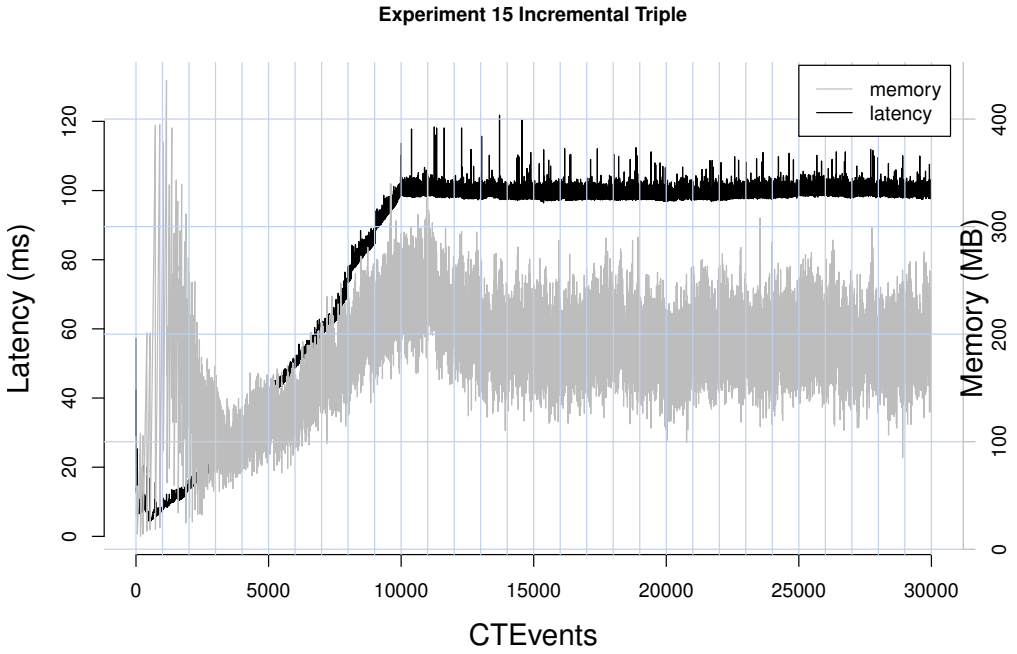


(b) GI Exp 12, CTEvent Size = 100, Num.Slot = 100

Figure 6.11 – Analyser Investigation Stack - Level 3 - Memory (Y) and latency (Y) behaviours in time domain (X) - SOAK Test Exp 7, GN, CTEvent Size = 10 N.Slot = 10 (a) and Exp 12, GI, CTEvent Size = 100 N.Slot = 100 (B).



(a) TN Exp 15, CTEvent Size = 1 Num.Slot = 10000



(b) TI Exp 15, CTEvent Size = 1 Num.Slot = 10000

Figure 6.12 – Analyser Investigation Stack - Level 3 - Memory (Y) and latency (Y) behaviours in time domain (X) - SOAK Test Exp 15, CTEvent Size = 1 Num.Slot = 10000, w.r.t baseline TN (a) and TI (b).

6.4 Step Response Test Evaluation Results

In the previous sections of this chapter we presented the three contributes that $\mathcal{H}$eaven provides to the SR research: (i) we explain how to design experiments through the $\mathcal{H}$eaven definition $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q} >$ (Section 6.1); (ii) we include two examples of test sets and the lecture-key to drive the result analysis for one of them (Section 6.2); (iii) we provide a concrete use-case analysis, exploiting one of the test set provided in (ii) (Section 6.3). However, in other to prove that $\mathcal{H}$eaven supports the SCRA we are still missing a forth contribution: (iv) proving the generality of (i), (ii), (iii) and consequently demonstrate the flexibility of the research powered by $\mathcal{H}$eaven.

Further examples of (i), (ii) and (iii) are not enough to show $\mathcal{H}$eaven capabilities proving (iv). We have to change the research paradigm, providing a complete different kind of analysis. SOAK Testing is a standard testing procedure for dynamic systems (see Section 2.5) and Step Response Testing is a standard complementary procedure for it. In order to prove the generality of the SCRA enabled by $\mathcal{H}$eaven, we provide an example of *Inter Experiment* analysis. We exploit the Step Response Test set presented in Section 6.2.2 together with the results of the SOAK Test set of Section 6.2.1, presenting a new kind of insight.

In the following we briefly show the Step Response testing results, exploiting the Analyser Investigation Stack, except for Level 2 as we explain in Section 6.4.4. We focus on the evaluation, because further details of each level of the Investigation Stack can be found in Section 4.3 and Section 5.4, while the design of this experimental set is described in Section 6.2.2.

6.4.1 Steady State Identification Block

The analysis starts with the identification of the Steady State condition for the variables involved in the experiments, memory and latency, as we did in Section 6.4.1. For SOAK Test experiments the Steady State condition has a single meaning w.r.t the execution. Instead, for Step Response Test experiments we can identify two possible Steady State conditions: ante step (AS) and post step (PS). In this section, we apply a different comparative approach,

contrasting the results of Step Response and SOAK experiment, which are relate as reported in Section 6.2.2.

The SSI block analysis is extended to reach this experimental characteristic. Tables 6.15 and 6.16 report the results from the SSI block. Each of them includes the results for the two phases which compose a Step Response Test: ante-step (AS) and post-step (PS).

As we expected, latency reaches Steady State condition for both the AS and the PS phases in all experiments. Instead memory does not reach the Steady State condition for several experiments, which are highlighted in Table 6.16, and independently from AS and PS phases.

CTEvent	CTEvent	GN		GI		TN		TI	
Init Size	Final Size	AS	PS	AS	PS	AS	PS	AS	PS
10	100	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
10	1000	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
100	1000	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 6.15 – Steady State Identification Block - Step Response Summary Table - The Steady State condition is evaluated for both the two phases of the Step Response Test execution: Ante Step (AS) and Post Step (PS). This tables report the SSI block results for the evaluation of the latency variable. All the experiment reach the Steady State for both the AS and the PS phases.

CTEvent	CTEvent	GN		GI		TN		TI	
Init Size	Final Size	AS	PS	AS	PS	AS	PS	AS	PS
10	100	Yes	No	No	No	Yes	No	Yes	No
10	1000	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
100	1000	No	Yes	No	Yes	No	Yes	No	Yes

Table 6.16 – Steady State Identification Block - Step Response Summary Table - The Steady State condition is evaluated for both the two phases of the Step Response Test execution: Ante Step (AS) and Post Step (PS). This tables report the SSI block results for the evaluation of the memory variable. Several experiments, highlighted in the table, do not reach the Steady State in one or both the AS and the PS phases.

6.4.2 Level 0 - Dashboard Views

In this section, we continue the evaluation providing some examples of an easy-to-ready dashboard view. As we did Section 6.2.1 where we tried to identify a

hierarchy between the tested solution (the Baselines).

The Level 0 comparison is summarised in two separated dashboards: Figure 6.13 shows the two phases of the Step Response Test execution, AS and PS. Figure 6.14 displays the results of the relative SOAK experiments. Considering the figures, SOAK I and AS results refer to a CTEvent size of 10, while SOAK F and PS refer to a CTEvent size of 1000.

Observing the figures we can read an insight: the performances at Steady State for both the AS and PS phases are similar to the relative SOAK one. This means that the transitory phase does not influence the Steady State condition reached for this particular experiment.

6.4.3 Level 1 - Statistical Values Comparison

In this section, we try to extend the statistical *Inter Experiment* comparison proposed in Section 6.3.3. We provide a different layout, which allows to compare the Baseline performances between the two phases of the Step Response tests and the relative SOAK experiment results.

Tables from 6.17 and 6.20 show the worsening of latency and memory values. We compare Step Response Test ante-step (AS) phase with the relative SOAK Test, named as SOAK I, and the Step Response post-step (PS) with the relative SOAK Test, named as SOAK F. We exploit the quantitative approach of Level 1 to evidence the differences between the two experiment results.

Tables 6.17 and 6.18 confirm what we have seen in the dashboard views at Level 0. We can observe that for the average values the variations between AS and SOAK I and between PS and SOAK F are minimal, around the 0.5% with some outliers of 10-12%. Exactly what the dashboards present in Figures 6.13 and 6.14.

Thus, the comparison of average data seems to be not so interesting. A better point of analysis is given by Tables 6.19 and 6.20, which apply the same layout to the comparison of memory and latency maximum values. Notice that we take all the execution, considering also the initial transitory phase which usually contains the most high values.

We compare the maximum latency values for the Test SOAK I and SOAK F, observing that: (i) the differences between SOAK I and AS, which involved the Step Response test warm-up, follow the same behaviour and confirms the

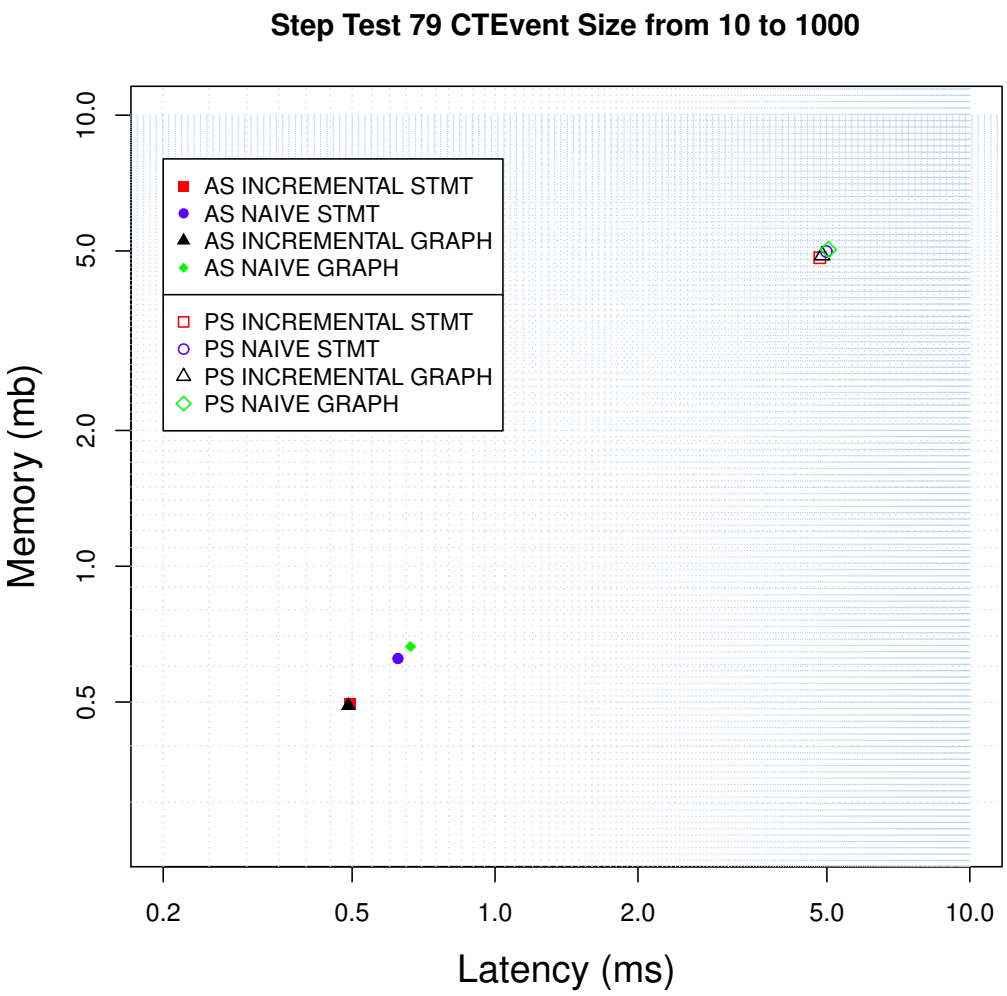


Figure 6.13 – Analyser Investigation Stack - Level 0 - Step Response Dashboard One - The figure shows in log-scale the average performance values for memory (Y) and latency (X) of the Step Response Test with an CTEvent Initial Size of 10 and with a CTEvent Final Size of 1000. We distinguish the two phases that compose the experiment as ante-step (AS) and post-step (PS) with different representation explained in the Legend

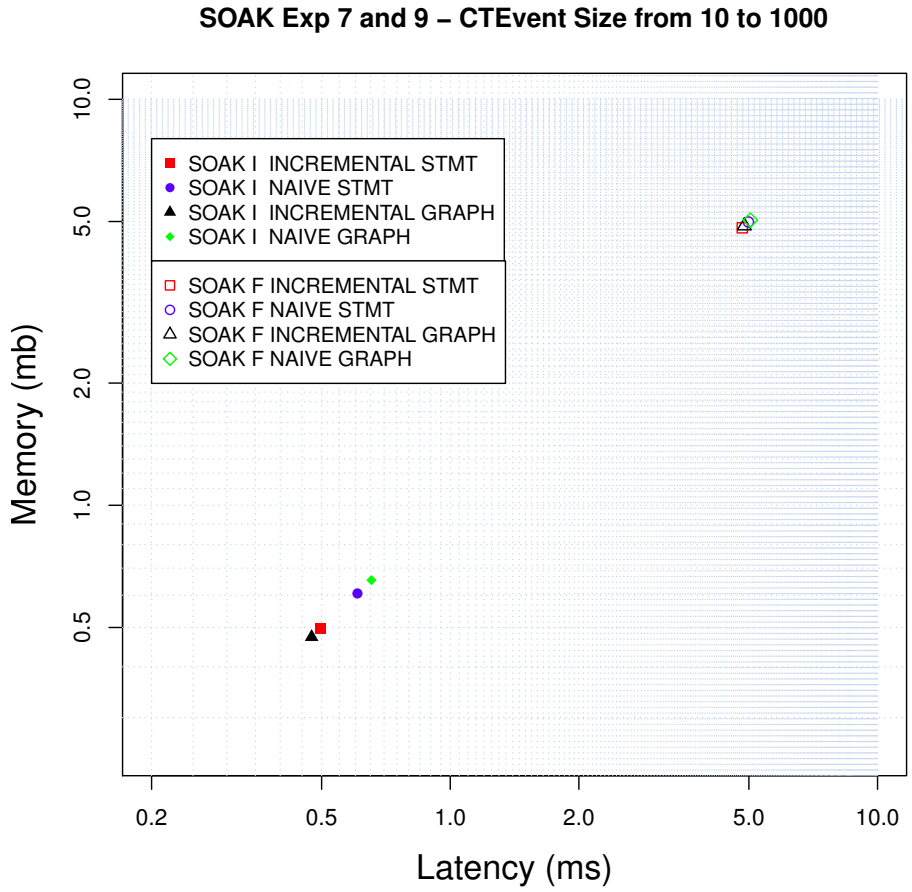


Figure 6.14 – Analyser Investigation Stack - Level 0 - Step Response Dashboard Two - Related SOAK Experiments - The figure shows in log-scale the average performance values for memory (Y) and latency (X) of the SOAK Experiment related to the Step Test of Figure 6.13. The SOAK test related to the first Step Response phase (SOAK I) has a CTEvent size of 10, while the SOAK test related to the second Step Response phase (SOAK F) has a CTEvent size of 1000.

insight from Table 6.17. (ii) The performance improvement between PS and SOAK F is much more relevant, always greater than 50% and some over the 80%. This insight points out that there is a initialisation cost that influences the performances, and it must be considered during the deployment of an RSP Engine.

Table 6.20 shows a different situation for memory. Nothing can be said about the comparison between SOAK I and Step Response AS except that the Step Response AS always consumes more memory w.r.t the SOAK I. Considering the comparison between Step Response PS and SOAK F, we cannot observe in general the same situation of Table Table 6.19, but a similar behaviour is present for Baselines GI and GN, which have a bigger increase in the Step Response PS than TN and TI.

6.4.4 Level 2 - Pattern Identification

As we reported in the introduction, we actually do not apply this level analysis. The motivation can be found in the restricted dimension of the Step Response test set, which lacks the minimum amount of information to perform pattern identification step. Moreover the number of slots, which compose the active window, is fixed to ten for this experiment set. In general DSMSs do not allow to change the window dimension during the execution. This limitation makes even less interesting the pattern analysis, since we have only a single controlling variable.

6.4.5 Level 3 - Single Visual Comparison

Finally, we conclude the SOAK Testing analysis through several examples of *Intra Experiment* comparison (see Section 6.3.5). In this section we provide the alternative analysis method of Level 3, presented in Section 4.3: the *Inter Experiment* comparison method.

Figures 6.15 (a) and (b) draw, on the same graphl, the latency and the memory series for three experiments. We build the comparison diversifying the *CTEvent* size, since the number of slots is fixed to ten.

Thus, the involved experiments are:

- Step Response Test with CTEvent initial size fixed to 10 Triples and Final Size fixed to 100 triples.
- SOAK Test Number 7, with CTEvent size fixed to 10 Triples.
- SOAK Test Number 8, with CTEvent size fixed to 100 Triples.

Notice that, Figure 6.15.a reports the latency time series for the Baseline TI, while 6.15.b presents the memory usage for the Baseline GI. We selected the two most expressive graphs, in order to show the following insight.

In Figure 6.15.a it is possible to see that after the step there is no transitory phase for the RSP Engine latency. A similar behaviour is observable, for memory, in Figure 6.15.b. About the Steady State condition, we can observe that latency behaviour does not changes if compare the SOAK Tests 7 and 8 with the relative Step Response phases AS and PS. The insight is quite more complex for memory. The growth trend is still present in both the SOAK and Step Response tests, but in the second one Java anticipates its optimisation policies, limiting the variance of memory oscillations.

Init Size	FinalSize	Graph Naive		Triple Naive		Graph Inc		Triple Inc	
		SOAK I	SOAK F	SOAK I	SOAK F	SOAK I	SOAK F	SOAK I	SOAK F
10	100	-0,83%	2,31%	-1,31%	1,42%	-2,12%	0,93%	0,12%	-0,49%
10	1000	-0,99%	0,32%	-1,80%	0,07%	-1,74%	-0,99%	0,12%	-0,56%
100	1000	0,47%	1,17%	-0,69%	0,00%	-0,93%	-1,06%	-1,54%	0,16%

Table 6.17 – Analyser Investigation Stack - Level 1 - Step Response Test average latency comparison.

Init Size	FinalSize	Graph Naive		Triple Naive		Graph Inc		Triple Inc	
		SOAK I	SOAK F	SOAK I	SOAK F	SOAK I	SOAK F	SOAK I	SOAK F
10	100	-9,50%	-12,04%	-1,39%	-0,44%	30,15%	-14,57%	-0,22%	-1,08%
10	1000	24,63%	2,28%	2,15%	-0,71%	1,35%	-9,39%	-5,89%	-0,52%
100	1000	1,96%	0,24%	0,16%	-0,39%	1,69%	-13,00%	0,62%	2,02%

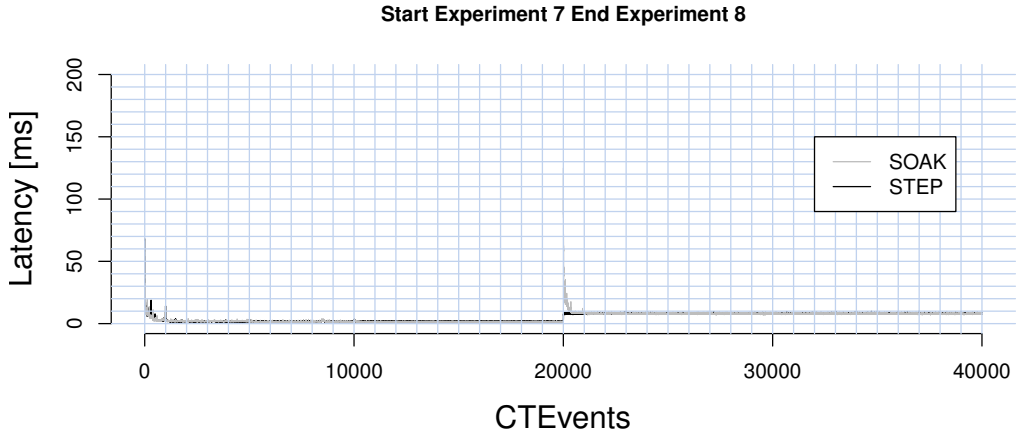
Table 6.18 – Analyser Investigation Stack - Level 1 - Step Response Test average memory comparison.

Init Size	FinalSize	Graph Naive		Triple Naive		Graph Inc		Triple Inc	
		SOAK I	SOAK F	SOAK I	SOAK F	SOAK I	SOAK F	SOAK I	SOAK F
10	100	-4,19%	86,86%	-8,82%	88,56%	-2,12%	91,67%	-3,03%	91,91%
10	1000	-3,14%	50,65%	-14,88%	54,64%	-1,74%	55,12%	-1,82%	61,78%
100	1000	-1,40%	51,53%	-6,61%	57,89%	-0,93%	53,80%	-2,40%	60,91%

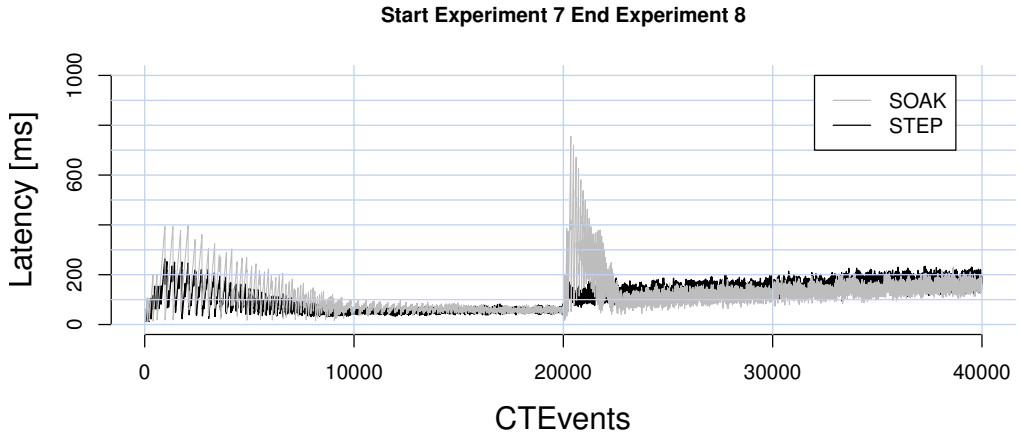
Table 6.19 – Analyser Investigation Stack - Level 1 Step Response Test maximum latency comparison.

Init Size	FinalSize	Graph Naive		Triple Naive		Graph Inc		Triple Inc	
		SOAK I	SOAK F	SOAK I	SOAK F	SOAK I	SOAK F	SOAK I	SOAK F
10	100	42,97%	57,59%	10,82%	25,56%	49,87%	67,48%	27,99%	17,64%
10	1000	39,42%	22,06%	17,25%	12,87%	25,12%	64,62%	37,69%	10,87%
100	1000	20,75%	29,71%	5,13%	12,43%	33,84%	69,09%	8,97%	27,48%

Table 6.20 – Analyser Investigation Stack - Level 1 Step Response Test maximum memory comparison.



(a) TI Step Response vs SOAK Tests - CTEvent Init Size = 10 Final Size 100 N.Slots = 10



(b) GI Step Response vs SOAK Tests - CTEvent Init Size = 10 Final Size 100 N.Slots = 10

Figure 6.15 – Analyser Investigation Stack - Level 3 - Inter Experiment comparisons of latency (Y) in Figure (a) and memory (Y) in Figure (b) over the all the experiment execution CTEvent (X).

Chapter 7

Conclusions and Future Works

In this thesis work, we presented $\mathcal{H}$eaven – an open source framework for empirical research of RSP Engines. $\mathcal{H}$eaven aims at enabling the Systematic Comparative Research Approach in the Stream Reasoning field, through an RSP Engine Test Stand, four Baselines of RSP Engines, and an Analyser.

The motivations that led this work are included in Chapter 3, while in Chapter 4 we described $\mathcal{H}$eaven design and in Chapter 5 we detailed the implementation of the Test Stand, the four Baselines and the Analyser. Finally, we presented an empirical proof of $\mathcal{H}$eaven potential in Chapter 6, providing examples of Experiment Design, two Test sets (SOAK and Step Response), and an evaluation of $\mathcal{H}$eaven Baselines, which exploits them.

We learnt that, even when RSP Engines are extremely simple (e.g., the baselines), it is hard to demonstrate hypothesis formulated only from a theoretical knowledge and, thus, empirical evaluation is required. Experiment results emphasised the importance of conducting comparative research based on controlled experimental conditions. Thus, we confirmed that the SR community needs an open source[1] framework like $\mathcal{H}$eaven.

The focus on the experimental infrastructure is the main difference between $\mathcal{H}$eaven and previous works. While SRbench and LSbench focus on RDF Streams and a suite of continuous SPARQL queries, $\mathcal{H}$eaven allows to compare RSP Engines based on any RDF Stream, ontology, continuous query and entailment regime. It even enables to run experiments connecting to live data streams as those used in [8].

[1] `https://github.com/streamreasoning/HeavenTeststand`

In this chapter, we recap this thesis works, presenting in Section 7.1 our Research Question and a brief description of $\mathcal{H}$eaven design and implementation, which answers such research question. Last but not least, in Section 7.2 we point out $\mathcal{H}$eaven limitations and the future works of this thesis.

7.1 Comparative Research of RSP Engines

Stream Processing research field is growing and the number of techniques to semantically handle data stream is increasing. RDF Stream Processing Engines, a.k.a. RSP Engines, are systems able to answer continuous extensions of SPARQL queries over RDF Streams. Due to their complexity, it is hard to systematic compare them under repeatable conditions,.

It is worth to note that, despite the Engineering epistemology of the Computer Science works, it is still present the lack of a Systematic Comparative Research Approach (SCRA) [46]. SCRA is typical of those research areas which have to face very complex systems, and have difficulty to simplify the models. Architectural analysis are useful, but they are not sufficient to evaluate RSP Engines, because their behaviour must be studied during the execution. For this reason, the Stream Reasoning community has tried to define and develop solutions to evaluates RSP Engines [44]. Recent works like [53, 41, 19] supported this approach with queries, dataset and methods. However the SR community still lacks an experimental infrastructure which enables the comparison of RSP Engines independently from RDF Stream, ontology, continuous query and entailment regime. From aerospace engineering we borrow the idea of engine test stand: a facility to develop engine through systematic testing under precise experimental conditions. Thus, we can formulate our research question as follow:

Can an engine test stand, together with queries, datasets and methods, support Systematic Comparative Research Approach for Stream Reasoning?

In the following we provide an evidence of how $\mathcal{H}$eaven positively answers the research question.

In Chapter 3, we describe how an engine test stand must be in the Stream Reasoning research field. We exploit the traditional experiment definition to

formulate the requirements that a Test Stand for RSP Engine must fulfil, in order to answer our research question and to grant the rigorous and systematic test of RSP Engines. SCRA, due to its case-oriented nature, demands simple terms of comparison, namely baselines, to exploit for initial evaluation examples. In Chapter 3, we detail which properties a baseline must have and we formulate them as requirements for their implementation.

In Chapter 4, we describe $\mathcal{H}$eaven design. We explain how the Test Stand and the Baselines should be to fulfil the requirements we posed. We introduce also the idea of the Analyser as an investigation stack that extends the research of RSP Engines from the traditional hypothesis based approach to the empirical and comparative one. In Chapter 5, we describe how $\mathcal{H}$eaven Test Stand and the Baselines are implemented. We also show how we realised the investigation stack, providing a statistical evaluation of experiment results at higher levels, while the lower ones offer an overview of the RSP Engine dynamics over all the experiment execution.

In Chapter 6, we show how the traditional top-down analysis are not enough for evaluating complex systems like RSP Engines, even in case of naive implementations. Our evaluation exploits an experimental set composed by SOAK Tests and Step Response Stress Tests, executed on the Baselines implementations that we included in $\mathcal{H}$eaven framework. The results of the analysis show how the traditional research, which formulate hypothesis only on the RSP Engine model knowledge, is still meaningful, but it can be improved through an infrastructure like $\mathcal{H}$eaven Test Stand. The evaluation conducted in Chapter 6 has shown that it is hard to demonstrate even naive hypothesis. RSP Engine dynamics can be only partially investigated from the statistical viewpoint. We need further knowledge about the RSP Engine dynamics, which means observing their behaviour at once and over the entire execution of an experiment.

$\mathcal{H}$eaven allows to drill down the analysis over an investigation stack which covers all the aspects of the dynamic system performance analysis. Through $\mathcal{H}$eaven is now possible to improve existing theoretical models thanks to the empirical findings that were not available before. Thus, we can positively answer our research question, stating that $\mathcal{H}$eaven sustains SCRA and extends the traditional top-down analysis.

7.2 Limitations And Future Works

During $\mathcal{H}$eaven development, we faced many issues related to the heterogeneous nature of RSP application domains. These concerns limit our work in different ways. They influence $\mathcal{H}$eaven development in term of both design and implementation. Moreover, our research of RSP Engines is actually restricted to $\mathcal{H}$eaven Baselines within an extremely controlled experimental setting.

The limitations on $\mathcal{H}$eaven design and its implementation must be faced, improving its models and further developing the current implementation of the Streamer, the ResultCollector and the Test Stand External Structure. On the other hand, the restrictions on the research of RSP Engines require to exploit $\mathcal{H}$eaven Test Stand in order to pursue the analysis. Finally, we consider a further possible contribution continuing the research on the Baselines, which has its own scientific value, as Chapter 6 partially evidenced.

Due to these limitations, the future works and possible extensions of $\mathcal{H}$eaven belong to the following categories:

- *Research of RSP Engine* - it involves the empirical evaluation of RSP Engine and the comparison of benchmarking results, which are our main research interests. Thus, we plan to support our research through $\mathcal{H}$eaven.
- *Software Engineering and Development* - it involves future works focus on the different aspects of $\mathcal{H}$eaven software, which is extendible by design.
- *Research on Baselines* - it aims to provide a complete evaluation of the Baselines as simple terms of comparison for mature RSP Engines.

We aim extending the *Research Work*, creating a ready-to-use benchmarking suite built upon $\mathcal{H}$eaven, which allows to test any RSP Engine with a set of well defined experiments. From preliminary studies we know that which is the essential test set to cover the most important uses cases.

An essential set of experiments must include the following tests:

- **T1** SOAK.
- **T2** Stress Step.

- **T3** Stress Sine Wave.
- **T4** Poisson Distribution.

The experiments definition still follows the tuple $< \mathcal{E}, \mathcal{D}, \mathcal{T}, \mathcal{Q} >$. The ready-to-use benchmarking suite will give the users the possibility to execute those test on their own RSP Engines. Moreover, it should include $\mathcal{H}$eaven Baselines as simple terms of comparison for benchmarking results.

SOAK Test [T1] and Stress Step Test [T2] are already part of this thesis work in a restricted form, while the other ones are not implemented yet.

In the current stage of development, it is possible to configure only the ontology and the entailment regime of the Baselines. Thus we develop [T1] and [T2] registering to our $\mathcal{E}$ as queries $\mathcal{Q}$ variations of the identity query, which differ for window size ω. We intend to continue the development of the Baselines, adding the possibility to register one or more continuous queries into them and exploiting more complex entailment regime than ρDF. Moreover, we have to define which queries $\mathcal{Q}$ to include in all the experiments, considering many works in the field [44, 53, 41, 19].

The current experiment sets [T1] and [T2] exploit LUBM ontology as $\mathcal{T}$ and the RDF Stream $\mathcal{D}$ is generated through a module, the RDF2RDFStream, which adapts LUBM data to a streaming scenario (See Chapter 5). In order to generate data for all the remaining test sets [T3] and [T4] we have to extend the RDF2RDFStream to generate: a random flow with a Poisson distribution, and a sine wave flow (to mimic the periodic changes in the flow rates observed on social media streams [8]).

Independently from the experiment set, we aim to extend the Test Stand measurement as suggested in [44]:

- Response time over all queries (Average/1^{th} Percentile/Maximum).
- Maximum input throughput in terms of number of data element in the input stream consumed by the system per time unit.
- Minimum latency to accuracy and minimum latency to completeness for all queries.

As we stated in Chapter 6, the content of the active window influences the RSP Engine performances for two factors: the window size before the

reasoning and after. The second metrics is relevant for the RSP Engine evaluation. When the system has to handle a big number of outgoing triples we can observe a degradation in terms of memory and latency. Thus, an evaluation of the number of inferred triples w.r.t the window content at time t allows to weight the engine performance results in relation to the input RDF Stream, helping to eliminate outliers and properly evaluate RSP Engines. Moreover, this observation opens new scenarios in the Stress Testing design, where the stress factor depends on the reasoning potential of the current window w.r.t a certain entailment regime.

Future works on *Software Engineering and Development* regard the Analyser. We aim to completely automate the analysis procedure, involving at least the current measurement set and tools.

As a long term goal, we intend to standardise the entire tool-set which supports analysis methods presented in Chapter 4. We are imagining the Analyser as a Web-based environment where all existing RSP Engines are already available; the experiments design and execution are accessible as a service through the selection of RDF Streams, ontologies, and queries; the results analysis of any external RSP Engine is allowed providing the data and the involved variables. Finally, a visual facility to compare different experiments and the publication of experiment result as linked data would complete this environment.

Last but not least, we would like to continue the *Research on Baselines*, because we identified an intrinsic scientific value in their evaluation, which can be another contribution itself. In order to study the problem of responsiveness we have to add four alternative implementations of the Baselines, which do not exploit the external time control. The Baselines should be evaluated by [T1], [T2], [T3] and [T4] but also with real data and a $\mathcal{T}$ different from LUBM, for example exploiting LS Bench queries and data to design an experiment set.

Bibliography

[1] Harith Alani, Lalana Kagal, Achille Fokoue, Paul T. Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha F. Noy, Chris Welty, and Krzysztof Janowicz, editors. *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, volume 8219 of *Lecture Notes in Computer Science*. Springer, 2013.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[3] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 480–491, 2004.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 1–16, 2002.

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 1–16, 2002.

[6] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[7] Marco Balduini, Irene Celino, Daniele Dell'Aglio, Emanuele Della Valle, Yi Huang, Tony Kyung-il Lee, Seon-Ho Kim, and Volker Tresp. BOTTARI: an augmented reality mobile application to deliver personalized and location-based recommendations by continuous analysis of social media streams. *J. Web Sem.*, 16:33–41, 2012.

[8] Marco Balduini, Emanuele Della Valle, Daniele Dell'Aglio, Mikalai Tsytsarau, Themis Palpanas, and Cristian Confalonieri. Social listening of city scale events using the streaming linked data framework. In Alani et al. [1], pages 1–16.

[9] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: a continuous query language for RDF data streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.

[10] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *SIGMOD Record*, 39(1):20–26, 2010.

[11] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying rdf streams with c-sparql. *SIGMOD Rec.*, 39(1):20–26, September 2010.

[12] Pedro Bizarro. Bicep - benchmarking complex event processing systems. In *Event Processing, 6.5. - 11.5.2007*, 2007.

[13] Jürgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking owl reasoners. In *Proc. of the ARea2008 Workshop, Tenerife, Spain (June 2008)*, 2008.

[14] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. *Proc. VLDB Endow.*, 3(1-2):232–243, September 2010.

[15] Jean-Paul Calbimonte, Óscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, pages 96–111, 2010.

[16] Jean-Paul Calbimonte, Hoyoung Jeung, Óscar Corcho, and Karl Aberer. Enabling query technologies for the semantic sensor web. *Int. J. Semantic Web Inf. Syst.*, 8(1):43–63, 2012.

[17] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[18] Emanuele Della Valle Daniele Dell'Aglio. Incremental reasoning on rdf streams. In Andreas Harth, Katja Hose, and Ralf Schenkel, editors, *Linked Data Management*, chapter 16, pages 413–436. CRC Press, 2014.

[19] Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Óscar Corcho, and Emanuele Della Valle. On correctness in RDF stream processor benchmarking. In Alani et al. [1], pages 326–342.

[20] Li Ding, Yun Peng, Paulo Pinheiro da Silva, and Deborah L. McGuinness. Tracking RDF Graph Provenance using RDF Molecules. Technical report, UMBC, April 2005.

[21] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[22] Joseph Felsenstein. Phylogenies and the comparative method. *The American Naturalist*, 125(1):pp. 1–15, 1985.

[23] Tom Gardiner, Ian Horrocks, and Dmitry Tsarkov. Automated benchmarking of description logic reasoners. 2006.

[24] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[25] Sven Groppe, Jinghua Groppe, Dirk Kukulenz, and Volker Linnemann. A SPARQL engine for streaming RDF data. In *Third International IEEE Conference on Signal-Image Technologies and Internet-Based System, SITIS 2007, Shanghai, China, December 16-18, 2007*, pages 167–174, 2007.

[26] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[27] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[28] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[29] IEEE. IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990.

[30] Thomas Janoski and Alexander M. Hicks. Methodological innovations in comparative political economy: an introduction. In *The Comparative Political Economy of the Welfare State*, pages 1–28. Cambridge University Press, 1994. Cambridge Books Online.

[31] Graham Klyne and Jeremy J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. World Wide Web Consortium Recommendation, February 2004.

[32] Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, pages 58–68, 2012.

[33] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proceedings of the 10th International*

Conference on The Semantic Web - Volume Part I, ISWC'11, pages 370–388, Berlin, Heidelberg, 2011. Springer-Verlag.

[34] Freddy Lécué, Simone Tallevi-Diotallevi, Jer Hayes, Robert Tucker, Veli Bicer, Marco Luca Sbodio, and Pierpaolo Tommasi. Smart traffic analytics in the semantic web with STAR-CITY: scenarios, system and lessons learned in dublin city. *J. Web Sem.*, 27:26–33, 2014.

[35] Chang Liu, Jacopo Urbani, and Guilin Qi. Efficient RDF stream reasoning with graphics processingunits (gpus). In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 343–344, 2014.

[36] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[37] Daniel A. Menasce and Virgilio Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[38] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

[39] Sergio Muñoz, Jorge Pérez, and Claudio Gutiérrez. Minimal deductive systems for RDF. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*, pages 53–67, 2007.

[40] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 345–355, 2000.

[41] Danh Le Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter A. Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines:

Facts and figures. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II*, pages 300–312, 2012.

[42] Eric PrudHommeaux, Andy Seaborne, et al. SPARQL query language for RDF, W3C recommendation, 2008.

[43] Yuan Ren and Jeff Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 831–836, 2011.

[44] Thomas Scharrenbach, Jacopo Urbani, Alessandro Margara, Emanuele Della Valle, and Abraham Bernstein. Seven commandments for benchmarking semantic flow processing systems. In *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings*, pages 305–319, 2013.

[45] Theda Skocpol and Margaret Somers. The uses of comparative history in macrosocial inquiry. *Comparative Studies in Society and History*, 22:174–197, 4 1980.

[46] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *J. Syst. Softw.*, 28(1):9–18, January 1995.

[47] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, 2003.

[48] Jacopo Urbani, Alessandro Margara, Ceriel J. H. Jacobs, Frank van Harmelen, and Henri E. Bal. Dynamite: Parallel materialization of dynamic RDF data. In *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, pages 657–672, 2013.

[49] Emanuele Della Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. A first step towards stream reasoning. In

Future Internet - FIS 2008, First Future Internet Symposium, FIS 2008, Vienna, Austria, September 29-30, 2008, Revised Selected Papers, pages 72–81, 2008.

[50] Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.

[51] Jacques Wainer, Claudia G. Novoa Barsottini, Danilo Lacerda, and Leandro Rodrigues Magalhães de Marco. Empirical evaluation in computer science research published by acm. *Inf. Softw. Technol.*, 51(6):1081–1085, June 2009.

[52] Onkar Walavalkar, Anupam Joshi, Tim Finin, and Yelena Yesha. Streaming Knowledge Bases. In *Proceedings of the Fourth International Workshop on Scalable Semantic Web knowledge Base Systems*, , Karlsruhe, DE, October 2008.

[53] Ying Zhang, Minh-Duc Pham, Óscar Corcho, and Jean-Paul Calbimonte. Srbench: A streaming RDF/SPARQL benchmark. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, pages 641–657, 2012.