# D2.1.2 Knowledge and Provenance modelling and Stream Reasoning v2

**Dr. Sina Samangooei, University of Southampton**
**David Monks, University of Southampton**
**Dr. Nicholas Gibbins, University of Southampton**
**Dr. Hans-Ulrich Krieger, Thierry Declerck, DFKI GmbH**

**Abstract.**
FP7-ICT Strategic Targeted Research Project (STREP) ICT-2011-287863 TrendMiner
Deliverable D2.1.2 (WP2)

**Keyword list**: knowledge modelling, stream reasoning

| | |
|---|---|
| **Project** | TrendMiner No. 287863 |
| **Delivery Date** | November 7, 2013 |
| **Contractual Date** | October 31, 2013 |
| **Nature** | Other |
| **Reviewed By** | Kalina Bontcheva, USFD and Alex Simov, ONTO |
| **Web links** | `https://github.com/sinjax/squall`; http://www.dfki.de/lt/onto/tmo.owl |
| **Dissemination** | PU |

Copyright © 2013 University of Southampton

## TrendMiner Consortium

This document is part of the TrendMiner research project (No. 287863), partially funded by the FP7-ICT Programme.

**DFKI GmbH**
Language Technology Lab
Stuhlsatzenhausweg 3
D-66123 Saarbrcken
Germany
Contact person: Thierry Declerck
E-mail: declerck@dfki.de

**University of Southampton**
Southampton SO17 1BJ
UK
Contact person: Mahensan Niranjan
E-mail: mn@ecs.soton.ac.uk

**Internet Memory Research**
45 ter rue de la Rvolution
F-93100 Montreuil
France
Contact person: France Lafarges
E-mail: contact@internetmemory.org

**Eurokleis S.R.L.**
Via Giorgio Baglivi, 3
Roma RM
00161 Italy
Contact person: Francesco Bellini
E-mail: info@eurokleis.com

**University of Sheffield**
Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Contact person: Kalina Bontcheva
E-mail: K.Bontcheva@dcs.shef.ac.uk

**Ontotext AD**
Polygraphia Office Center fl.4,
47A Tsarigradsko Shosse,
Sofia 1504, Bulgaria
Contact person: Atanas Kiryakov
E-mail: naso@sirma.bg

**Sora Ogris and Hofinger GmbH**
Linke Wienzeile 246
A-1150 Wien
Austria
Contact person: Christoph Hofinger
E-mail: ch@sora.at

**Hardik Fintrade Pvt Ltd.**
227, Shree Ram Cloth Market,
Opposite Manilal Mansion,
Revdi Bazar, Ahmedabad 380002
India
Contact person: Suresh Aswani
E-mail: m.aswani@hardikgroup.com

# Executive Summary

This deliverable is about knowledge modelling and stream reasoning in the context of the TrendMiner project. The document is organized following those two topics. We start with the actual state of development of the TrendMiner Ontologies (TMO), and then describe the actual state of development of stream reasoning.
The integrated TrendMiner Ontologies have been built partly from scratch and consist of existing but also of updated ontologies. The need for a set of TrendMiner specific ontologies for pursuing the task of Ontology-Based Information Extraction (OBIE), which is the topic of Task 2.3 in TrendMiner, has been recognized very soon, since for the specific use cases described in WP6 and WP7, one can not rely only on the generic ontologies available for example in DBpedia or Freebase.
To address the increasing number of high throughput semantic streaming data sources, we present the Squall distributed stream reasoner. Using Squall, rules and queries can be instantiated as production systems which can consume unbounded streams of data, producing meaningful application-specific answers to structured questions. We present two versions of Squall. In the first version delivered, a complete tool and framework are described which can instantiate and realise Jena rules and SPARQL queries. In the second version a more modular version of the Squall framework is delivered capable of expressing: different query and rule languages; production system optimisation strategies; and instantiations on different stream processing frameworks.

# Contents


**1 Introduction** **3**
- 1.1 Relevance to Trendminer . . . 4
  - 1.1.1 Relevance to project objectives . . . 4
  - 1.1.2 Relevance to other work packages . . . 4
- 1.2 Software Availability . . . 4

**2 TrendMiner Ontologies** **6**
- 2.1 Introduction . . . 6
- 2.2 Ontologies . . . 6
  - 2.2.1 *BIO* . . . 8
  - 2.2.2 *EN* . . . 9
  - 2.2.3 *ICB* . . . 9
  - 2.2.4 *OP* . . . 10
  - 2.2.5 *IF* . . . 10
- 2.3 Rules . . . 12
  - 2.3.1 Finding Competitors Across Stock Exchanges . . . 12
  - 2.3.2 Monitoring Unusual Events . . . 12
- 2.4 Relevance for Multilingual Lexical Resources . . . 13

**3 The Squall stream reasoner** **14**
- 3.1 Introduction . . . 14
- 3.2 Stream Reasoning . . . 14
- 3.3 Squall Overview . . . 16
- 3.4 Squall Internals . . . 18
  - 3.4.1 Streams and Distribution . . . 18
  - 3.4.2 Query Processing . . . 19
  - 3.4.3 Reasoning . . . 27
- 3.5 Conclusion . . . 28

**4 Modular Stream Reasoning** **29**
- 4.1 Introduction . . . 29
- 4.2 Modular Production Rule Systems . . . 31
  - 4.2.1 Lexing . . . 31

4.2.2 Translators . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2.3 Planners . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.2.4 Builders . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3.1 Jena Translator . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3.2 Greedy Planner . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.3.3 OpenIMAJ Stream Builder . . . . . . . . . . . . . . . . . . . 38
4.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

**5 Conclusions and Future Work 41**
5.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2.1 Retraction Support . . . . . . . . . . . . . . . . . . . . . . . . 42
5.2.2 Ontology Translation . . . . . . . . . . . . . . . . . . . . . . . 42

# Chapter 1

# Introduction

We describe two activities in the context of Task 2.2 of TrendMiner: Knowledge Modelling and Stream Reasoning.

Knowledge Modelling in TrendMiner is needed in order to pursue the task of Ontology-Based Information Extraction (OBIE, task 2.3, see also D2.2.2 Multilingual, ontology-based IE from stream media -v2). There ontological data available in the Linked Open Data framework is already used, mainly for the purpose of Named Entities disambiguation. But the generic ontological data one can find in for example DBPedia, is not suited for providing the specific knowledge background for the use cases in TrendMiner, which are dealing with *assisting financial investing decisions* (WP6) and *EU-wide tracking of political views, trends, and politician popularity over time* (WP7).

We have therefore constructed an integrated ontology, **TMO**, the TRENDMINER Ontology, that has been assembled from several independent multilingual taxonomies and ontologies which are brought together by an interface specification, expressed in OWL (McGuinness and van Harmelen (2004)). The TMO ontologies are online (http://www.dfki.de/lt/onto/tmo.owl)
As Semantic Web technologies have increased in maturity, they have been increasing applied to application domains that may be characterised by the large volumes of data that they generate and the real-time demands on the processing of that data. Examples of such domains include sensor networks, financial information and communications network management. Over the last decade, the database community has shown a great deal of interest in Data Stream Management Systems (DSMS), which aim to meet the demands of such domains. However, although stream processing systems have been developed to deal with streaming data, there are only a few systems which have attempted to do so for streaming semantic data. Moreover, the current generation of stream processing systems for semantic data typically leverage existing reasoners and query engines by applying them to snapshots of the streams, rather than adopt a native stream processing approach.

This document accompanies both the TMO delivery and the software deliverable of the Squall stream reasoning tool that has been built as part of Work Package 2, and which

follows the initial sketch that was provided in Deliverable D.2.1.1. The structure of this document is as follows: in Chapter 3, we describe the basic Squall system, which implements a production rule system for RDF using the Rete algorithm on the Storm stream processing framework; in Chapter 4 , we describe the enhanced Squall system, which decomposes the task of building the Rete network into modular components, so allowing the specification of rules in a wider variety of languages (Jena, RIF, SPARQL) and on a wider variety of processing frameworks (Storm, S4, Hadoop).

In the remainder of this chapter, we describe the relevance of this deliverable both to the project objectives, as given in the Description of Work, and the the research taking place in other Work Packages.

## 1.1 Relevance to Trendminer

### 1.1.1 Relevance to project objectives

The TMO ontologies are relevant for all work packages that need to have access to stable knowledge objects (instances).

The streaming reasoning tool described in this deliverable contributes both to project objective 1 (”deliver new real-time trend mining and summarisation methods for stream media”) and objective 5 (”deploy a cloud-based infrastructure for real-time collection, analysis, summarisation, and semantic search”).

### 1.1.2 Relevance to other work packages

The TMO ontologies are for sure relevant for all OBIE activities, described in WP2, but is also supports greatly the use cases, and is also relevant to WP4 on summarization.

The stream reasoning tool contributes principally to the real-time stream media platform that is being developed in Work Package 5, but it also acts as an upstream component of the machine learning tools developed in Work Package 2; the entailments that are generated by the stream reasoner can be viewed as extra features that can be used to help identify important messages or trend information.

## 1.2 Software Availability

The schema of the TMO is available online: www.dfki.de/lt/onto/tmo.owl.

All software including the source for the various modules and methods by which the tools

can be built are found on the project's Github repository[1]. The README of the project contains instructions by which the project and its components can be included as part of a mavenised java project, a development environment can be created, or a version of the tool can be compiled and run.

[1] `https://github.com/sinjax/squall`

# Chapter 2

# TrendMiner Ontologies

## 2.1 Introduction

Within TRENDMINER, TMO serves as a common language that helps to interlink data, delivered from both symbolic and statistical components of the TRENDMINER system.

Very often, the extracted data is supplied as *quintuples*, RDF triples that are "annotated" by two further temporal arguments, expressing the temporal extent in which an atemporal fact holds (essentially, an extension of the plain N-Triples format; see http://www.w3.org/TR/rdf-testcases/). In order to store such quintuples, they are either transformed into a set of semantic-preserving triples when stored in a triple repository like OWLIM (

In this paper, we will also sneak a peek on the temporal entailment rules (Krieger (2012)) and queries that are built into the semantic repository hosting the data and which can be used to derive useful explicit information. This includes identifying companies operating in similar areas, monitoring data for unusual events, or making knowledge about people explicit.

## 2.2 Ontologies

Overall, TMO consists of sixteen truly independent ontologies which do not have knowledge of one another. Two further ontologies, called *IF* and *XEBR2XBRL* bring them together through the use of interface axioms, using axiom constructors, such as rdfs:subClassOf and owl:equivalentProperty, or by posing domain and range restrictions on certain underspecified properties. It is worth noting that across the ontologies, each property has been cross-classified as being either *synchronic*, i.e., property instances staying constant over time, or *diachronic*, i.e., changing over time (Krieger (2010)). This property characteristic can be used, amongst other things, to check the consistency of a

temporal ABox or as a distinguishing mark in an entailment rule.

Let us quickly introduce the **18 sub-ontologies of TMO** and then focus on a few selected **highlights** in subsections 2.1–2.5.

1. *BIO* (biographical facts about people and events)
2. *CFI* (ISO's classification of financial instruments)
3. *DAX* (stock exchange: Deutscher Aktien Index)
4. *DC* (very less from Dublin Core)
5. *EN* (stock exchange: NYSE Euronext)
6. *GICS* (Standard&Poor's/MSCI industry sector classification)
7. *ICB* (Dow Jones/FTSE industry sector classification)
8. *IF* (most of the interface axioms)
9. *LOGIC* (modalized propositions; used by *SENT*)
10. *NACE* (EU/UN industry sector classification)
11. *OP* (opinion: extends the MARL ontology)
12. *POL* (political facts about people and events)
13. *SENT* (sentiment, uses *LOGIC*)
14. *SKOS* (SKOS relations applicable to classes)
15. *SOC* (translation of TheSoz/GESIS sociology thesaurus)
16. *TIME* (distinction: synchronic/diachronic properties)
17. *XEBR* (XBRL Europe Business Registers)
18. *XEBR2XBRL* (interfacing XEBR and local XBRL jurisdictions)

Even though ABox data (populated instances) usually come with a temporal extent, the TBoxes and RBoxes of the ontologies are not equipped with temporal information, thus still being represented as triples. For instance, we do not state that an URI represents a class at a certain time and a property at a different time. Or that a class is a subclass of another class for only some amount of time. Thus TBox and RBox of the integrated ontology represent universal knowledge that is true at any time, so there is no need to equip them with a fourth and fifth temporal argument. This quality gives rise to the use of ontology editors such as Protégé for manually constructing the TBoxes and RBoxes of some of our ontologies.

It is worth noting that almost all ontologies are multilingual in that both classes, properties, and predefined instances are assigned multiple and multilingual labels or

even longer definitions in different languages, making use of the annotation properties rdfs:label, skos:prefLabel, and skos:altLabel, together with an additional annotation property rdfs:definition.

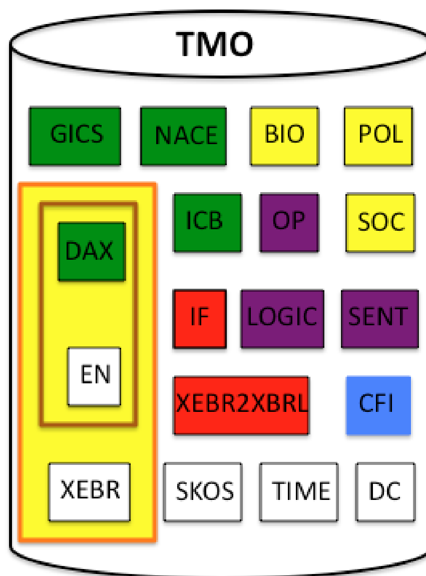


Figure 2.1: The TMO ontology consists of 18 sub-ontologies overall. The color encoding refers to ontologies focussing on models of *people's private and public life* (**yellow**), *sentiment/opinion* (**purple**), *industry sector classification* (**green**), *stock exchange* (**brown rectangle**), *financial reporting* (**orange rectangle**), *financial instruments* (**blue**), and *interface* (**red**). As can be seen from the picture, some of the ontologies even model several aspects of a domain; e.g., *DAX* alone deals with industry sector classification, reporting, description of stock exchange listed information, and people who are either key executives or shareholders of a company.

### 2.2.1 *BIO*

*BIO* is used to represent biographical facts about people's life. The ontology comes with a tripartite structure of the following pairwise disjoint classes, subclasses of the most general class Entity:

- Abstract. An Abstract thing can manifest in a Happening, whereas a Happening might lead to an Entity.
- Happening. Happenings are either static atomic Situations or dynamic decomposable Events.
- Object. The subclass Agent makes a fundamental distinction between Person and Group.

Assertional knowledge (i.e., ABox relation instances) in *BIO* is usually encoded as quintuple (see above) with the notable exception of Happenings which encapsulate their own starting and ending time. People might be involved in a happening or learn about it, thus being isAwareOf a happening. Since isAwareOf is a diachronic property, we can easily model (using the temporal extent) that awareness might turn into oblivion.

### 2.2.2 *EN*

The Euronext ontology *EN* does not come up with its own industry classification, but makes use of *ICB* (see below). *EN*, in part, reduplicates the stock exchange ontology *DAX*, but uses different names for classes and properties. However, more financial numbers are given here, even for three succeeding year.

Let us take an example. *Credit Agricole*'s revenues for 2012, 2011, and 2010 are listed today as 16,315,000, 20,783,000, and 20,129,000 Euros. Even at the end of this year, these numbers will be the same. However in 2014, the number for 2010 will no longer be listed, but instead, we will then find only numbers for 2013, 2012, and 2011. Clearly, we do *not* want to extend the ontology with new property names every time a new business year starts, thus we must avoid properties such as hasRevenue2013, hasRevenue2012, hasRevenue2011, etc.

In order to address this and to proper represent the numbers against the varying date when the numbers where taken, we use a simple "trick" here: we always use exactly the three properties hasRevenue-1, hasRevenue-2, and hasRevenue-3. The hyphen - now should be interpret as a minus sign $-$, thus, e.g., the value stored under hasRevenue-2 actually refers to the revenue *two* (2) years ago *relative* to the actual business year when the company snapshot was taken (the business year is stated elsewhere).

### 2.2.3 *ICB*

ICB, the *Industry Classification Benchmark*, is a further industry classification used worldwide at many places (e.g., NYSE in New York). Euronext (as operated by NYSE) makes use of ICB's four level deep classification in its description of titles. Given the ICB terminology stated in a document, we have auto-generated an OWL ontology *ICB* that arranges the 186 industry sectors in a subsumption hierarchy. *ICB* is connected to *EN* through an axiom from *IF* and comes up with an informal sector description for English, German, and Spanish, together with a further multilingual "definition" of the most specific concepts (actually, even 11 languages are available). In order to address these definitions on the class level properly, we make use of a further annotation property which we have called rdfs:definition.

### 2.2.4 *OP*

This opinion ontology is based on the *Marl* ontology, described in (Westerski et al. (2011)). Even though some of the property names would have been labelled differently by us (e.g., using hasTarget instead of describesObject in order to be compatible with opinion mining terminology), we have not alter the original property names.

We have, however, made some adjustments to *Marl* and have added further properties as described below:

- extractedFrom is now a datatype property, mapping to xsd:anyURI;
- we have added the object property hasHolder (range: underspecified);
- we have added the datatype property holdersTrust (range: xsd:double);
- we have added the datatype property utteredAt (range: xsd:dateTime);
- we have declared certain properties to be functional;
- we have defined the range type for already-existing properties.

Some of the original properties (e.g., describeFeature) as well the new property hasHolder are not assigned a range class in *Marl*. In order to constrain these properties further, we recommend (as we have done in TMO) to add further interface axioms, e.g., *the holder of an opinion is an agent/person* (see subsection 2.2.5 below).

We have furthermore classified all properties in the opinion ontology as diachronic properties. This has the advantage that such a treatment makes it easy to see how an opinion evolves/changes over time. Note that this evolution mostly happens for aggregated opinions, but might even happen for information related to a single opinion, say the holders' trust changes.

### 2.2.5 *IF*

As already explained, the interface ontology *IF* interlinks the 16 subontologies through manually specified interface axioms. To achieve this, *IF* makes use of *DC*, *SKOS* and *TIME*, but mostly utilizes the standard axioms constructors from RDFS and OWL, together with domain and range restrictions:

- owl:equivalentClass
- rdfs:subClassOf
- owl:equivalentProperty
- rdfs:subPropertyOf
- owl:sameAs

- rdfs:domain
- rdfs:range
- rdf:type

Here are some examples, using description logics (DL) syntax.

#### Classes and Properties

dax:Company, en:Company, nace:IndustrySector, and gics:GICS can be used interchangeably; xebr:Report is a subclass of dc:Resource; the properties dax:portrait and en:activity are equivalent (DL syntax):

$\text{dax:Company} \equiv \text{en:Company}$

$\text{dax:Company} \equiv \text{gics:GICS}$

$\text{dax:Company} \equiv \text{nace:IndustrySector}$

$\text{xebr:Report} \sqsubseteq \text{dc:Resource}$

$\text{dax:portrait} \equiv \text{en:activity}$

Note that the transitivity of owl:equivalentClass guarantees that dax:Company, en:Company, gics:GICS, and nace:IndustrySector are lying in the same equivalence class.

#### Domain & Range Restrictions and Typing

XEBR reports are linked to companies via the diachronic functional object property if:hasReport; the holder of an opinion is an agent:

$\top \sqsubseteq \forall \text{if:hasReport}^{-} . \text{dax:Company}$

$\top \sqsubseteq \forall \text{if:hasReport} . \text{xebr:Report}$

$\text{if:hasReport} : \text{owl:FunctionalProperty}$

$\text{if:hasReport} : \text{owl:ObjectProperty}$

$\text{if:hasReport} : \text{time:DiachronicProperty}$

$\top \sqsubseteq \forall \text{op:hasHolder} . \text{bio:Agent}$

The last axiom together with

$\text{bio:Person} \equiv \text{pol:Person}$

gives us the possibility to talk about, e.g., journalists and their opinions, due to the following subclass axioms, specified in *BIO* and *POL*, resp.:

$\text{bio:Person} \sqsubseteq \text{bio:Agent}$

$\text{pol:Journalist} \sqsubseteq \text{pol:Person}$

## 2.3 Rules

This section presents some showcases that involve individual ontologies, interlinking axioms, and domain-specific queries and entailment rules.

### 2.3.1 Finding Competitors Across Stock Exchanges

Characterizing a company against an industry sector classification is an extremely important showcase which involves finding competitors of a company that work in a similar field. We attack this problem in two ways. Firstly, we have established manual mappings between sectors from different classification schemes, such as (all four classes talk about *financial institutions*)

icb:ICB8300 $\equiv$ nace:nace_64.1

icb:ICB8300 $\equiv$ dax:Banks

icb:ICB8300 $\equiv$ GICS4010

Secondly, we are trying to match the free-text information of a company against the multilingual labels of the NACE, ICB, DAX, and GICS classes in order to establish an automated sector classification. For instance, from the English info text found for *adidas*

*The adidas Group is one of the global leaders within the*

*sporting goods industry ...*

it should be feasible to find the class nace:nace_47.64 whose English label is

*Retail sale of sporting equipment in specialised stores*.

Since the mappings connect industry sectors across different stock exchanges, querying for companies of type dax:Banks will automatically yield companies classified as icb:ICB8300, nace:nace_64.1, or GICS4010. Here is an example involving competitors of *Deutsche Bank*, making use of the query language in HFC (Krieger (2013)) to access quintuples in the `WHERE` clauses:

```
SELECT DISTINCT ?competitor
WHERE ?db dax:name "Deutsche Bank" ?s ?e
      ?db rdf:type ?type ?s ?e
      ?competitor rdf:type ?type ?s2 ?e2
FILTER ?db != ?competitor
```

### 2.3.2 Monitoring Unusual Events

"Unusual" events refer to important changes happened in a company or in a person's life, say, the replacement of a CEO or the change of the *transparency standard* (a company

can not adhere to more than one standard at the same time). If latter happens, a rule can leave a *memento* in the repository that can be queried later. Here is an example, making use of HFC's rule language:

```
?c dax:transparencyStandard ?ts1 ?s1 ?e1
?c dax:transparencyStandard ?ts2 ?s2 ?e2
->
?mem rdf:type if:Memento ?e1 ?s2
?mem if:changeStandard ?c ?ts1 ?ts2 ?e1 ?s2
@test
?ts1 != ?ts2
DTLess ?s1 ?s2
@action
?mem = MakeUri ?c ?e1 ?s2 ?ts1 ?ts2
```

The predicate (`@test`) `DTLess` guarantees that `?s1` is smaller than `?s2` (both variables will bind XSD atoms of type `dateTime`). The action (`@action`) `MakeUri` deterministically generates a new URI from its input arguments `?c`, `?e1`, `?s2`, `?ts1`, and `?ts2`. This URI then is used on the RHS of the rule to store the relevant information, viz., the company, the different standards, and the period in which the change happened.

## 2.4 Relevance for Multilingual Lexical Resources

In general, linguistically-analyzed multilingual language data used in labels, comments, and definitions of knowledge organization systems (KOSs) can be a very rich input for multi-lingual Ontology-Based Information Extraction (Wimalasuriya and Dou (2010)), ontology mapping, and translation (Montiel-Ponsoda et al. (2011); Garcia et al. (2012)), or for multi- and crosslingual terminology harmonization across various KOSs (Gromann and Declerck (2012)). As such, language data in KOSs are building a specific but very useful language resource, since their encoding in RDF (using for example the *lemon* model (McCrae and Unger (2014))) is a way to explicitly link language data with domain knowledge. Due to space requirements, we can not go into further details of those aspects and have mainly focussed on the description of the integrated ontologies. In the final version, we will address the multilingual aspects of our ontologies in more detail.

# Chapter 3

# The Squall stream reasoner

## 3.1 Introduction

In this chapter, we present a novel approach to reasoning over streaming semantic data that implements the reasoner as a data flow network, and a prototype implementation of that approach, Squall. The stream-native approach taken in Squall differs from current approaches, which leverage existing Semantic Web reasoners. We therefore begin this chapter with a review of the literature on stream reasoning, and then examine the design of the Squall reasoner in more detail.

## 3.2 Stream Reasoning

Traditional database management systems (DBMS) adopt a 'store now, query later' approach in which largely static data is organised in a persistent data set. This is appropriate for an application where the data will be queried repeatedly, and where updates to the data set are small or infrequent; consequently, DBMSes typically provide indices or other access structures that improve the efficiency of access to the stored data.

By contrast, many recent applications deal with large volumes of constantly changing dynamic data; these applications are characterised by update event frequencies often in excess of 100,000 events per second, and appear in a variety of domains from online auctions and financial trading systems to sensor networks and social media. The assumptions made by DBMSes, that the amortised cost of building access structures are far outweighed by the savings to be made by using those access structures, no longer hold with such applications, and so different approaches are needed. Moreover, the data rates for some applications are high enough that they may preclude the persistent storage of data in any way.

In the past decade, the database systems community has explored the area of *data stream*

*management systems* (DSMS), special purpose databases which are designed to handle unbounded sequences of time-varying data. These systems are characterised by the use of *continuous queries*, long-lived queries that are matched against streaming data as it is received (as opposed to the one-shot queries that are typical in DBMSes), and by an explicit acceptance of the tradeoff between cost and completeness of results.

Developments in data storage on the Semantic Web over the same period have predominantly focussed on *triplestores* - DBMSes designed for storing RDF triples - and it is only in the last few years that any significant attention has been paid to *streaming semantic data* (della Valle et al., 2009).

Although work on continuous queries dates back at least as far as the early 1990s (Terry et al., 1992), concerted work on data stream management systems did not start in earnest until a decade later. Babcock et al. (2002) gives a concise survey of the early DSMSes. A common feature of these early systems share is that they adopt a data flow approach to the processing of incoming data and to the evaluation of continuous queries. Queries are decomposed into fundamental operators that are arranged as the vertices in a directed acyclic graph (the query plan - which may combine multiple queries), the edges of which correspond to the data streams that are the inputs and outputs of those operators. This allows the operators to be executed independently of each other, increasing the overall flexibility of the system.

Streaming semantic data is a recent development in the Semantic Web community that applies the techniques used in DSMSes to RDF data, and represents a radically different approach to that more usually found in the Semantic Web. The popularity of *linked data* is at least in part due to the assumption of persistence of data (the notion that ”cool URIs don’t change”), whereas streaming data is by its very nature fleeting and ephemeral. This is in many ways a natural development for the Semantic Web; in the real world, data exists at all points on the spectrum from persistent to ephemeral. Moreover, the application of a DSMS approach to the Semantic Web can be seen as the continuation of a long-standing flow of techniques from the databases community. For example, the development of efficient RDF triple stores has been possible principally because the Semantic Web community has been able to build extensively on forty years of research into relational databases; the semantics of the SPARQL query language (Perez et al., 2009) rely in no small part on the relational algebra.

The transition from DSMSes that operate on streams of arbitrary tuples, to DSMSes that are restricted to operate on RDF triples/quads is therefore a relatively straightforward move that several research groups have made largely independently of each other. Notable examples of such semantic stream processing are the streaming SPARQL work of Bolles et al. (2008), the C-SPARQL query language of Barbieri et al. (2009) and the EP-SPARQL query language of Anicic et al. (2011); these languages typically extend the semantics of SPARQL by defining an RDF stream as a set of timestamped subject-predicate-object triples, mostly following the approach made in the STREAM DSMS (Arasu et al., 2003) (with some minor changes; both Bolles and Anicic annotate

triples with time intervals, rather than the instants used by STREAM).

These semantic stream processing approaches may be extended to *stream reasoning*, whereby streams of entailments may be generated from streams of RDF data.One of the earliest references to stream reasoning is by della Valle et al. (2008), who present two conceptual architectures for combining reasoning techniques with data streams. The first of these architectures is based on RDF molecules and reuses existing DSMSes and reasoners by coupling them using a transcoder that converts from the format used by the existing DSMS to timestamped RDF molecules, and a pre-reasoner that incrementally maintains materialised RDF snapshots. These snapshots are passed to conventional SW reasoners that are not aware of time. This may be combined with the incremental materialisation algorithm described by Barbieri et al. (2010) which maintains the ontological entailments. This algorithm, based on the *delete and re-derive* approach introduced by Gupta et al. (1993), tags each RDF triple (both inserted and entailed) with an expiration timestamp and applies a sliding window to the stream of timestamped triples. The algorithm then can compute a new complete and correct materialisation by dropping RDF triples that are no longer in the window – effectively temporal truth maintenance.

The second architecture in (della Valle et al., 2008) primarily concerns itself with querying rather than reasoning *per se*, and streams RDF triples (rather than molecules); here, stream operators (like those in the STREAM DSMS) are arranged in query plans. Other similar contemporary approaches include that by Walavalkar et al. (2008), who use the rule-based axiomatisation of RDF Schema to generate a set of continuous queries to be evaluated within the TelegraphQC DSMS (Chandrasekaran et al., 2003), and that by Hoeksema and Kotoulas (2011), who arrange a set of S4 processing elements, whose functionality corresponds to the RDFS entailment rules, to generate a stream of inferred triples which is then fed back into the reasoner (in order to calculate the deductive closure of the stream).

## 3.3 Squall Overview

The Squall reasoner is a novel stream reasoner that can evaluate both continuous queries in C-SPARQL, and collections of rules. As part of Deliverable D2.1.2, we have provided two versions of Squall; in this chapter we describe the basic version, while in Chapter 4 we describe a modular extension to Squall. Squall depends on a number of open source technologies, including the Apache Jena RDF library[1] and the Storm system for distributed real-time computation[2]. Squall's novelty lies in two areas: it reasons natively on streams, and that reasoning may be distributed across a server cluster.

By native stream reasoning, we mean that, rather than reusing an existing reasoner, as in the RDF molecule stream reasoning approach described above, we have taken an approach which combines several aspects of the systems described in Section 3.2, specif-

[1] `http://jena.apache.org/`

[2] `http://storm-project.net/`

ically the fine-grained streaming of (della Valle et al., 2008), the rule-based nature of (Walavalkar et al., 2008) and the re-entrant streams of (Hoeksema and Kotoulas, 2011). The system evaluates rules and queries by translating them into data flow networks that apply low-level operators (such as join and select) directly to the data streams, rather than by applying a conventional reasoner to a snapshot of the data streams.

Key to this approach is the development of an efficient query plan that corresponds to the queries and entailment rules that are in use. In the basic Squall system, we use the well-understood Rete pattern matching algorithm (Forgy, 1979) to transform the rule bodies into a query plan. The Rete algorithm has long been used to improve the efficiency of matching facts against productions in production systems, and is in essence a data flow system; partial matches are propagated through a network of alpha and beta nodes (effectively select and join operators) in such as way as to minimise the number of times that each new fact is matched against a pattern. In adapting the Rete algorithm to streams of RDF triples, the alpha and beta memories become respectively streams and windows on streams (beta memories appearing immediately before beta or join nodes); this is effectively the approach taken by Jin et al. (2005) in the ARGUS stream processing system.

In order to support entailment rules beyond those in the axiomatisation of the ontology language, the basic version of Squall accepts Datalog-style rules expressed in the rule language used by Jena (the enhanced version of Squall, described in Chapter 4, supports the BLD dialect of the Rule Interchange Format (Boley and Kifer, 2010) in addition). These rules are compiled to a Rete-based query plan and deployed such that the output stream from the rule network (the merge of the streams resulting from the terminal nodes that represent the head of each rule) can be fed back into the network in order to calculate the deductive closure of the entailment rules, as in (Hoeksema and Kotoulas, 2011) (but note that the entailment stream is not reentrant by default). The output stream can also be fed into a second Rete network, this time for the continuous queries that have been registered with the system. This partitioning of the Rete network simplifies its management; the rule network is relatively static (the entailment rules are expected to persist for the lifetime of the system), whereas the query network is more dynamic (although long-lived, the continuous queries do not necessarily persist as do the entailment rules). An outline sketch of the reasoner from a data flow perspective is shown in Figure 3.1.

In order to reduce the amount of duplicate processing, the query plan built for the rules allows the sharing of operators between different rules (so, for example, if two rules contained the same pattern in their head, only a single select node matching that pattern would be present in the resulting rules). However, there is no structure sharing between the rule query plan and the plans built for the continuous queries that have been registered with the system, nor is there any sharing between different continuous query plans; this decision follows the observation made above regarding the expected lifetime of rules versus queries, and aims to minimise the amount of changes that need to be made to the query plans.

Having established the query plans, the nodes of each are then provisioned as a Storm

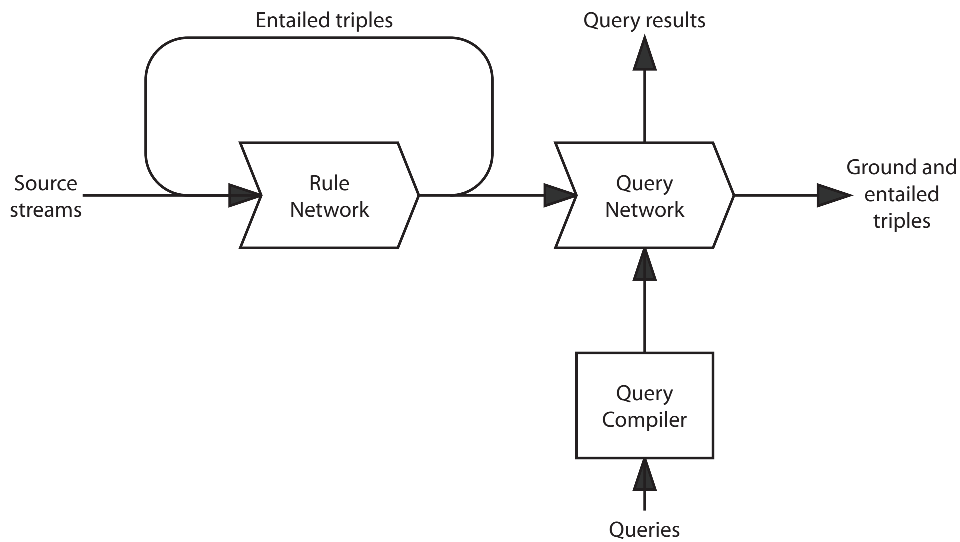


Figure 3.1: Stream Reasoner data flow

topology, which allows the processing of the plan to be distributed; the nature of this distribution is left to the Storm framework.

## 3.4 Squall Internals

In this section, we take a more detailed look at key aspects of the basic Squall system, concentrating on the underlying stream representation, and the processing of queries and rules.

### 3.4.1 Streams and Distribution

The streams of RDF triples that pass through the Squall system are managed via the Storm framework, typically using Kestrel message queues[3]. Streaming data sources add new triples to an input queue, which the Storm framework wraps up as a *spout* (a source of streams). The atomic unit of processing within Storm is the *bolt*, a node which processes any number of input streams and produces any number of new output streams. Many functions may be implemented as bolts; in Squall, we primarily concentrate on filters and

[3] http://robey.github.io/kestrel/

joins. The spouts and bolts within a system are organised as a *topology*, a network which performs an arbitrarily complex multi-stage stream computation.

### 3.4.2 Query Processing

In this section we define how we compile a Storm topology from C-SPARQL queries and Jena rules in Squall topologies. We outline our overall compilation technique including a discussion of the technologies we use and extend. We go on to describe how the various components of distributable queries can be implemented within the Storm framework.

We use the primitive Storm components of spouts and bolts to compile a topology designed to answer a specific C-SPARQL query. The spouts in a Squall topology emit graphs of triples as opposed to individual triples. In most streaming contexts, a group of triples which relate to a given event are more likely to arrive simultaneously than individual independent triples. For example, a single social media event (tweet, facebook post etc.) is unlikely to be a single triple, but instead a graph of triples defining the event's various attributes. Beyond spouts, all components in the Squall topology consume some input and emit bindings. For example: the filter bolts match specific triple patterns on inputs of graphs and emit bindings set to components of matched triples; join bolts compare the value of shared bindings of two other bolts and emit all bindings of two triples which match a shared binding; SPARQL Filter bolts decide whether a specific binding of variables passes a given filter statement and forward the input bindings on if so. With this in mind, in the next section we describe in detail how each component takes its input and returns bindings of variables.

During the compilation of C-SPARQL queries into Storm topologies, we have opted to implement within-query structure sharing. Conceptually, if two bolts have an identical history and binding variable configuration they can be treated as the same bolt and therefore any function they perform can be performed once on a given stream and its emitted bindings used multiple times. For example, the SPARQL query defined in Figure 3.2 contains two basic graph patterns and could therefore be intuitively defined using two bolts which feed a single join. However, if one disregards the specific variables *?user1*, *?user2* and *?friendOfFriend* these two bolts are identical[4]. A filter defined on either basic pattern would match the same triples and emit the same bindings as the other. How these bindings are joined together, defined later in the topology compilation process, would then represent the two patterns we see in the query. With the careful design of binding order we reuse simple filter bolts as well as arbitrary trees of join bolts allowing for increased efficiency within queries. We also hope to investigate this strategy as an approach to pattern reuse across queries. We describe how re-use is implemented in the various topology components below.

[4] i.e. they are both of the format ? `foaf:knows ?`

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?user1 ?friendOfFriend
WHERE {
    ?user1 foaf:knows ?user2 .
    ?user2 foaf:knows ?friendOfFriend
}
```

Figure 3.2: Within-query shared structure

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
SELECT ?user1 ?friendOfFriend
FROM STREAM <http://thesocialnetwork.com/posts>
            [RANGE 30m STEP 5m]
FROM STREAM <http://yourspace.com/walls>
            [RANGE 1h]
WHERE {
    {
        ?user1 foaf:knows ?user2 .
        ?user2 foaf:knows ?friendOfFriend
    }
    {
        ?user2 dc:created ?post
        ?post dc:createdAt ?time
        FILTER (time > "20/01/2012"^^xsd:date)
    }
}
```

Figure 3.3: Example C-SPARQL query

### Stream Definition

In C-SPARQL, the “FROM STREAM” construct as shown in Figure 3.3 must contain an IRI from which triples can be physically downloaded: “...the IRI represents an IP address and a port for accessing streaming data...” (Barbieri et al., 2009). The exact approach by which triples are extracted from these resolvable streams can take many forms and there are various implementation issues to consider, especially in a distributed context. For example, a given stream might be used by multiple queries simultaneously running on the same cluster making it intuitively more efficient to download triples a single time from the stream and make them accessible by all topologies. In this setup, an IRI would act more as an index into a local register of streams rather than a first-class source of triples.

With regards to distribution, extremely high volume streams might benefit from physical distribution around the cluster as opposed a to a single point of entry, both for purposes of resource utilisation but also so that the single point of entry doesn't become a point of failure in the distributed stream processing system. Further complication arises when the input streams are consumed by multiple distributed sources, including the difficulty in guaranteeing that a specific input is consumed only once and, if consumed, processed successfully.

To address these concerns, Squall supports many different kinds of stream implemented as various kinds of spout. A *simple stream spout* is a single-task Storm component which makes a direct connection to the IRI. This spout is useful for development and testing or for low throughput streams and local RDF files. In anything beyond this, this spout would represent a single point of failure and result in inefficient network utilisation in larger scale streams. If such a simple spout were distributed, each node would consume the IRI simultaneously and have no way to guarantee that a message wasn't a duplicate of another already served by another spout instance. Therefore Squall also supports a more complex message queue-based triple spout. As mentioned above, we use the Kestrel distributed message queue system. A Kestrel distributed message queue cluster involves a set of Kestrel servers, each of which holds a reliable ordered message queue. By picking a random kestrel server for all get operations (i.e. when the spout emits) and set operation (i.e. when the kestrel queue is filled from the IRI) one loses the ordering of messages, but gains a highly distributable source of triples while allowing both for guarantees of no processing duplication but also (if required) processing guarantees.

Though we discuss these specific examples, many spouts can be written for arbitrary sources of triples. The key feature all Squall spouts share is their ability to create serialised Jena Graph instances from an IRI and emit these graphs with a timestamp to subscribing bolts.

### Basic Graph Patterns

In Squall, the lowest granularity level of SPARQL component represented by a single physical component on a Storm topology is the SPARQL *basic graph pattern* which we represent as *filter bolts*. These bolts hold a single triple pattern which may contain zero or more variables which must be bound by incoming triples. When receiving the serialised Graph instances from the spouts, the filter bolts use their basic graph pattern to query the Graph through the Jena API, receiving zero or more triples which match the literals of the triple pattern and provide bindings for the variables of the triple pattern.

During query compilation, the variables of the basic graph pattern are numbered in order of appearance within the graph pattern[5]. This order is used to designate the position of a binding in the tuples emitted by the filter bolt. This approach provides a variable-name agnostic method for variable binding which allows a single filter bolt to be shared

[5] e.g. `?user foaf:knows ?friend` becomes `?0 foaf:knows ?1`

between multiple basic graph patterns which share a given triple pattern, but which may not share exact variable names within. These variable names are forgotten at run time and in fact only come into operation in the final terminal bolt and when join bolts are being constructed. We describe this process in more detail in the next section.

### Joins and Groups

In Squall, joins represent a point at which a pair of sources[6] have their variable bindings compared to one another. When two triples from the two joining sources have equal values bound to variables which the two sources are being joined upon, they emit a novel set of variable bindings integrating the variables they shared, but also the bound values for the variables they did not share. This is implemented using two queuing data structures representing sliding windows over the two sources being joined. When a novel binding appears from the left or right source, the corresponding queue is offered the binding. At this point the sliding windows are updated, removing any triples that have fallen out of the window either by time considerations or space. Simultaneously to being added to the queue of origin, the binding is compared to each binding in the sibling queue. If the bound values for the variables on which the two sources are being joined match then the values of the two bindings, (i.e. both the matching variables and the associated variables which were not involved in the join) are combined and emitted, thus joining the results of the two sources.

During the compilation phase, the variable names involved in the bindings of the two components are matched against one another such that the index of the matching bindings from both the left and right is noted in each join bolt. For example, when joining `?user1 foaf:knows ?user2` and `?user2 foaf:knows ?friend` it would be noted that the second variable on the left must match the first variable on the right. With this variable-match-index information, a pair of bindings from two sources is identified as matching if-and-only-if each value bound to each binding index for the two sources are equal. This approach of variable-index matching rather than direct variable-name matching allows bindings to be matched against each other at run time in a binding variable-name agnostic fashion. From this it follows that if two join instances have the same incoming pattern on the left and right and both emit the same pattern, they can share a single join bolt instance.

At this stage it is valuable to discuss groups as defined by SPARQL queries. It is possible to consider the group graph pattern construct (i.e. anything contained within a pair of curly brackets) to represent items which must be preferentially joined. Without the presence of *FILTER*, *UNION*, *subqueries* or *OPTIONAL* statements, such groups elements could correctly[7] be joined in any arbitrary order. For example: the query pattern $\{A \bowtie B\} \bowtie \{C \bowtie D\}$ expresses the same query as $\{A \bowtie B \bowtie C \bowtie D\}$ or indeed $\{A \bowtie B \bowtie C\} \bowtie D$. There is in fact a great deal of opportunity in the space of join op-

[6] either filters or indeed other joins

[7] According to the SPARQL 1.1 specification

timisation when selecting precisely how to join such ambiguous joins. This is a particularly important consideration in a distributed streaming context due to the existence of sliding windows. One join configuration could mean a constantly overflowing window which loses a lot of bindings while another could result in early filtering of irrelevant bindings and thus less window overflowing. In Squall we purposefully avoid this optimisation and instead chose to take the group graph patterns defined by the query as the query's join plan. The single optimisation we do apply is that, at a group level, if the join order is ambiguous we preferentially join bolts which share variable names in an attempt to avoid joins which emit bindings for every pair of input bindings[8]. For exmaple, the query pattern $\{A \bowtie B \bowtie C \bowtie D\}$ is constructed in Squall as though it were $\{\{\{A \bowtie B\} \bowtie C\} \bowtie D\}$ assuming the clauses $A$, $B$, $C$ and $D$ shared no variables[9] but would be joined $\{\{\{A \bowtie D\} \bowtie B \bowtie\}C\}$ if $A$ and $D$ shared variables.

Finally, we must discuss exactly how we make joins work correctly in Storm and therefore a distributed context. Our goal is not only to gain distribution by spreading different joins across a network, but to distribute the operations of an individual join, spreading the load of the binding/match operation described above across a cluster of machines and therefore achieving higher throughput. However, to distribute a specific join operation we must guarantee that a specific join bolt *task* is guaranteed to be given all instances of bindings which could ever match each other, while simultaneously not defaulting to a situation where all bindings are sent to a single task instance, which would guarantee correct matches but would not distribute load. To these ends we use the fields grouping provided by Storm, which allows for aggregation of tuples based on the values they hold. Concretely, each join bolt registers with its left and right sources using a fields grouping on the indexes of the fields which are involved in the join between the two sources. This guarantees that bindings from the left and right sources are distributed across the cluster while simultaneously guaranteeing that bindings which have the potential to match each other are never sent to different bolt task instances.

### Filters

In SPARQL, filters have group level scope which means that any bindings within the filter's group, or groups held recursively inside the filter's group, must pass their bindings to the filter in order to be considered as valid bindings and pass down through the rest of the query. In a Squall topology, for a filter to be given access to all the bindings it needs, the filter must occur as the last step in the network path of given group. This principal is reflected by the SPARQL standard which states that $\{A \bowtie B \bowtie FILTER(?a == ?b)\}$ is equivalent to $\{A \bowtie FILTER(?a == ?b) \bowtie B\}$. Subsequently, at compile time, each filter discovered within a group is held out until all other structures within the group are constructed (i.e. basic graph pattern bolts are connected in a network of join bolts), at which point a SPARQL filter bolt per FILTER expression within the group is connected to

[8] This is the case when a join is against two branches which share no variable names

[9] A rather strange query to be sure but useful for this discussion

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc:   <http://purl.org/dc/elements/1.1/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
SELECT ?friend ?post
FROM STREAM <http://thesocialnetwork.com/posts>
            [RANGE 30m STEP 5m]
FROM STREAM <http://yourspace.com/walls>
            [RANGE 1h]
WHERE {
    ?post dc:createdBy ?friend
    {
        ?user1 foaf:knows ?friend .
    }
    UNION
    {
        ?user1 foaf:knows ?user2
        ?user2 foaf:knows ?friend
    }
    ?user1 foaf:mailBox "ss@ecs.soton.ac.uk"
}
```

Figure 3.4: An example C-SPARQL query supported by Squall

the final join bolt in a chain of filters, stopping bindings from flowing between a grouping and components further up the query hierarchy if they do not pass the SPARQL filter's binding requirements.

As filters act only as a validation of a set of bindings in SPARQL, they hold no windows, instead only being handed complete sets of bindings to check. Therefore, it is safe to connect filters as shuffle groupings in Storm. In terms of internal implementation, each filter node holds a single SPARQL *FILTER* clause which is used to construct a Jena *ElementFilter* instance. By using the Jena framework to implemented SPARQL filters in Squall we immediately support most valid SPARQL filters in a streaming context.

### Unions and Optional

Operationally, a UNION statement can be thought of as a fork in the Squall topology, fundamentally constructing two completely separate networks at the point of the UNION. Consider the query in Figure 3.4 which requests the posts made by people known by, or people known by people who are known by *?user1*. The first basic graph pattern of *createdBy* and can be expressed as a simple filter bolt. If joined in the order the constructs appear in the query, the union is then expressed as a fork from that filter bolt. The first

path in the fork results in the *createdBy* filter joining with the first option in the union, the second path in the fork results in its join with the second part of the union. These two seperate join paths are then joined individually and seperately with the final compoenent of the query, namely the *mailBox* filter.

This approach still allows for shared bolts. In Figure 3.4 there are in fact 3 unique filter bolts, one for the *createdBy*, *mailBox* and *knows* components. These filters are then joined and indeed the join between *createdBy* and *foafKnows* is shared by the first and second union paths. Our forking approach means unions are represented as separate paths of tuple flow, while simultaneously allowing for shared components as these paths could happen to go through shared bolts.

The implementation of OPTIONAL has a conceptually similar implementation to UNION in Squall. Fundamentally a fork of two paths is constructed such that one path has the OPTIONAL component, and the other path does not. These forks created by both OPTIONAL and UNION are maintained in the topology until the final terminal at which point the bindings are combined to answer the specific query.

An optimisation might be an early merging of forks which would allow the reuse of whole union patterns rather than a component-wise reuse. This optimisation is reserved for future work. Note however, that without this optimisation, the UNION is a construct which doesn't explicitly exist at runtime. Instead the forks are only considered during compilation time and are held implicitly by the structure of the topology at runtime.

### Aggregation

An important feature of SPARQL which has vital application in a streaming context is aggregation. Indeed, the main purpose of queries over a streaming data source is often some summary statistic over a window of the stream to provide some real time understanding of the values in the stream. The implementation of the various aggregation operators is handled in the final terminal bolt. If aggregation is requested, a configurable window of bindings are held and the aggregation is calculated over them. The aggregation is emitted and calculated, including stale values in the window removed, every time a new binding is handed to the final terminal. We use the Jena implementation of the SPARQL aggregation operators to support aggregation in Squall the first instance. However, the Jena implementations do not support the updating of statistics, instead relying on a batch update of all statistics every time the window is updated. There are various approaches to the efficient implementation of aggregation operators over a stream, we leave these optimisations for future work.

The nature of aggregation requested in a given query decides exactly how the final terminal is to be connected to the rest of the Squall topology. Firstly, in the case where no aggregation is requested, the final terminal can be heavily distributed and connected to the rest of the network through a shuffle grouping. This is the case only because without aggregation each binding is in and of itself the answer to the SPARQL query and can

be emitted in isolation of other bindings, this is the most efficient form of final terminal connection. The next most efficient case is when aggregation is requested together with a GROUP BY against some variable bindings. In this case, a fieldGrouping can be used such that bindings containing like-values for the variables contained in the GROUP BY statement are all sent to the same final terminal bolt tasks. In this case the final terminal can still be distributed, however in an extreme case where the GROUP BY variables take only a one or few values the network may be underutilised. Finally, in the worst case of aggregation without GROUP BY (e.g. *SELECT count*$(*)$...) all bindings must be sent to one and only one running final terminal bolt task. In this case the answer to aggregation can be accurate only if a single bolt task handles all bindings, otherwise the requested summary statistic would be only a partial view of the aggregation. Therefore in this scenario a global grouping is used to guarantee a correct aggregation.

### Terminals

The final bolt type is the terminal bolt, which takes the role of combining bindings to answer the specific SPARQL query. The main role of the final terminal bolt is the transformation of bindings into query answers, and the transmission of query answers to some external output. Similar to spouts, this final terminal could support many different outputs depending on the task at hand. In the current implementation of Squall a file output is supported, useful for tests and benchmarks, and a Kestrel queue output is supported which is useful for larger scale production usage.

In a query which does not contain any unions or optionals there are three configurations a network can take. The network can be a single bolt (i.e. a single filter) and therefore connect directly to the conflict set. In the second configuration the network may be a set of filters combined through a set of join bolts, in which case all joins will be eventually pairwise joined with each other until a single final join is reached, this bolt is then connected with the final terminal. Finally, in the third configuration there may be a set of filters in the highest level group, in this case once bindings pass the chain of filters they are passed directly to the final terminal. Therefore, in all cases not involving a union a single bolt will simply connect directly with the final terminal. In the case of a union the final terminal bolt is simply connected to each of the final single bolts of each of the union forks.

In any case, when bindings are pass to the conflict set bolt the results of the query are constructed. Squall currently supports SELECT and CONSTRUCT SPARQL queries, though ASK is trivial to support. SELECT takes all bindings given and outputs the requested bindings. If any individual variable has no value in a given binding the only conclusion which can be reached is that the other variables were a part of an OPTIONAL part of the query pattern and therefore are left blank in the output. In any other case the binding could not have reached the final terminal. These bindings can be outputted in various formats including: RDF, JSON or CSV. The output of a CONSTRUCT follows a similar process to SELECT but instead of the output being variable bindings, the output

is formed from the requested triples template filled with the requested variable bindings.

The distribution of the final terminal and how the final terminal is connected to it's preceding bolts, is decided by the level of aggregation requested in the given query.

### 3.4.3 Reasoning

In Section 3.4.2 above, we have described the processing of queries within the basic Squall system. The evaluation of rules uses the same techniques; alpha nodes within the Rete are implemented as filter bolts, while beta nodes are implemented as joins. The alpha memories which lie between the alpha and beta networks are implemented using the message queues underlying the Storm framework, as are the beta memories which lie between beta nodes.

The key differences between our treatment of continuous queries and our treatment of rules are as follows:

1. Each continuous query added to the system is deployed as a separate Storm topology; there is no sharing of nodes between queries. In contrast, each collection of rules is compiled to a single Storm topology, even though the bodies of the rules may be of similar complexity to typical queries, which allows a great deal of node sharing within each collection of rules. The construction of the rule topologies uses the Rete algorithm laid out by Forgy (1979).

2. The terminal bolts in the rule topologies emit triples, rather than the bindings emitted by those in the query topologies. In a query topology, the *final terminal bolt* takes on an analogous role to the conflict resolution phase and the production execution phase of the Rete algorithm; each such bolt consists of a set of triple patterns which are instantiated by the incoming bindings to produce the triples that result from a successful activation of the underlying rule.

3. The stream of triples that are emitted from the terminal bolts may be combined (using a bolt which subscribes to each terminal bolt) and piped back into the input stream of the reasoner, as shown in Figure 3.1. This reentrancy allows the (repeated) application of the rules to their own entailments, in order to produce more of the deductive closure of the input stream (limited by the window sizes placed on the input - if this window size were infinite, the reasoner would be complete, so this permits a degree of control in the tradeoff between completeness and resource usage)

## 3.5 Conclusion

The last decade has seen a radical increase in the number and throughput of semantic streaming data sources. These data sources, though information rich, are fundamentally useless without systems capable of reasoning over streaming context and querying structured semantic streams over time. In this chapter we have outlined the squall stream reasoning library and tool. Given rules written in Jena or structured queries written in CSPARQL, squall can construct production systems capable of reasoning over semantic data streams. The production systems constructed by squall allow for node sharing implementing the rete production system algorithm. Squall instantiates these production systems over storm allowing for distributed reasoning and horizontal scaling in order to handle extremely high throughput streaming data sources.

# Chapter 4

# Modular Stream Reasoning

## 4.1 Introduction

For the first version of the tool described in the previous section, a concerted effort was made to make provided software modular and extensible. Therefore, it was possible to write a SPARQL query production system which could be extended to deal with a variety of sources and outputs as well as extensions to provide for static data sources. However, the extensibility of the code fell short when it came to sharing the structure of a SPARQL production system with a production system made from a different source, for example, Jena rules. It is known for example that the basics of the rete production rule system can be as readily applied to answering SPARQL queries as it can be applied to Jena rule sets.

Further, the system discussed in the previous section makes no allowance for different physical instantiations of the production rule system. A given rule language is consumed and a physical instantiation of the system is made directly, in our case in the Storm distributed computing framework. However, if in future the same production system was to be deployed against yahoo's S4 distributed stream processing framework, or perhaps against a non-streaming batch framework, it would be very difficult to reuse the components of the system which do not relate to the physical manifestation of the system, such as the optimized query plan..

Towards a modular production rule instantiation we highlight 4 separable stages of the production rule construction process. These stages are: *(1)* Lexing, *(2)* Translation, *(3)* Planning and *(4)* Building .

During the lexing stage the proposed production system is consumed from its source language. Examples of this process are the consumption of Jena rules or the consumption of SPARQL queries. This stage is relatively simple and often provided by off-the-shelf lexers for a given production rule language.

The output of the lexer does not necessarily guarantee a direct mapping between different production rule systems. At the translation stage a lexed production rule system is taken

from its raw form and expressed as a series of translated components. The translator must classify all the production rule components as one of a fixed set of translated components as well as defining a function which actually performs the task of each distinct production rule component of the lexed production rule system.

After the translators have been applied, the production rule systems is expressed in uniform way called a Translated Production System (TPS). The next stage is a process of solution Planning wherein a TPS is expressed in a form of a Planned Production System (PPS) which can be built directly. The planners make no attempt to understand the granular functionality of the translated components within the TPS. Instead, using only the component's type and input/output variable bindings which TPS components must provide, planners express how the components should be connected to one another in order to best instantiate a given production rule system. More concretely the output of the Planner is a Directed Graph (digraph). The vertices of the this digraph contain the functionality of the components of the TPS. In simple planners there may be a one to one mapping of components to nodes. In more complex planners single PPS nodes might contain the functionality of multiple TPS components. Further, the planner explicitly defines how these components must be connected to one another using a series of directed edges. A given planner works independently of the preceding translator which means that the optimisations achieved by a certain planner can be applied as readily to solving Jena rule production systems as it can be applied to a SPARQL production rule system.

A planner provides the optimised structure of a production rule system as a digraph. To finally instantiate a given production rule system, a builder must be defined. The job of the builder is to express the digraph described by the planner against a particular communication and processing framework. The internal workings of the components and the data transmitted is of no consequence to the Builder. Builders must only concern themselves with the practical details of how given production rule nodes connected by edges in a digraph are instantiated and that provisions are made for the communications between these nodes.

By using a combination of these 4 components, production rule systems can be instantiated which are written in arbitrary languages (e.g. Jena, RIF, SPARQL), optimized using arbitrary production rule system algorithms (e.g. RETE, Eddys etc.) and finally expressed against arbitrary communication frameworks (e.g. Storm, S4, Hadoop). We provide this framework as well as at least one working implementations of each of the 4 components in the Squall[1] library. In the rest of this section we describe each of these 4 components in more detail as well as describing the provided implementations.

[1] `http://github.com/sinjax/squall`

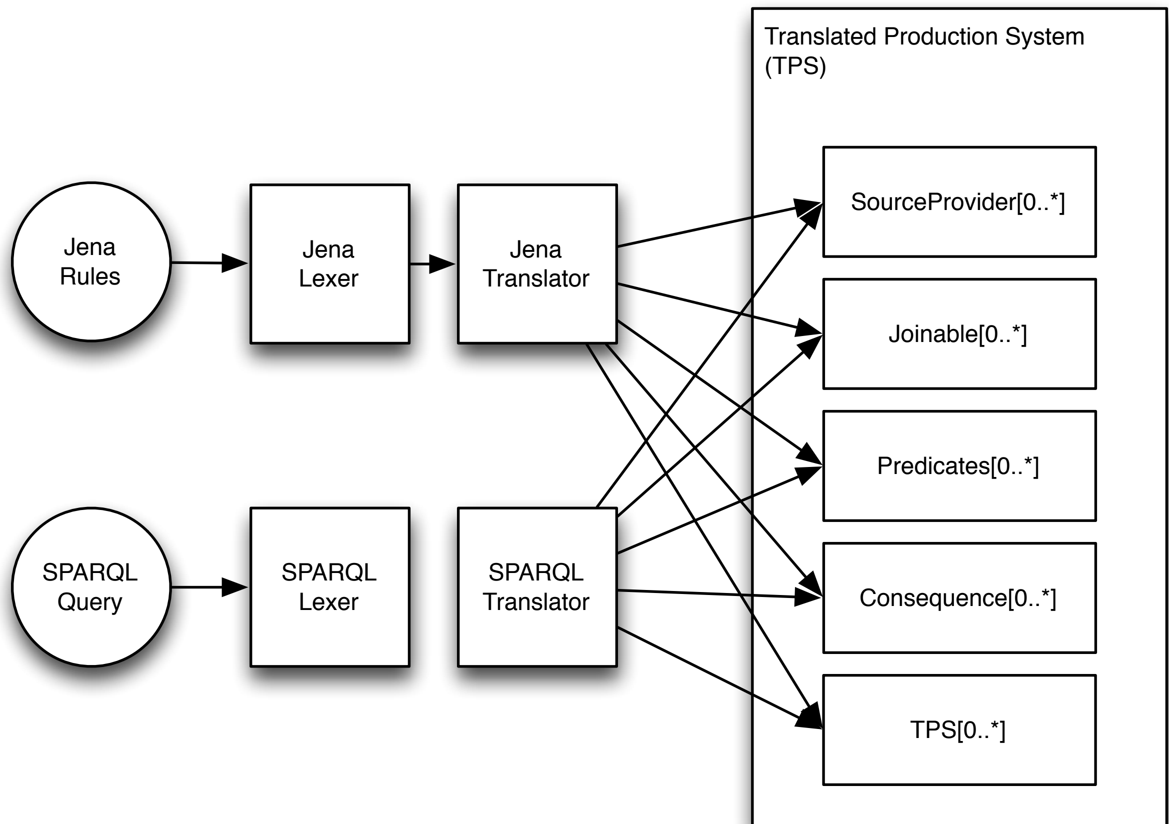


Figure 4.1: Multiple lexer and translator sets for different production system definitions, all outputting to a unified translated production system (TPS)

## 4.2 Modular Production Rule Systems

### 4.2.1 Lexing

Individual Lexers are often black box solutions understood by a specific Translator. This is in direct contrast to the *translator* $\rightarrow$ *planner* and the *planner* $\rightarrow$ *builder* relationships which can be combined arbitrarily. In Squall we use the predefined SPARQL and Jena Rules lexers made available in Jena. We have also written our own lexer for the consumption of RIF rule sets.

### 4.2.2 Translators

Coupled with an appropriate lexer, a translator defines a given production rule system as a set of components classified as one of a predefined set of component types. The goal of a

Translator is the transformation of a raw production system definition as a Translated Production System (TPS). Generally, the components of a TPS are functions which consume some data (bindings, triples etc.), and output some data (bindings of variables, triples etc.). More concretely, an individual component is often expressed as a function which consumes a *Context* and outputs a *Context*. A Context in Squall is an OpenIMAJ streaming construct which is in essence a map of strings to arbitrary objects. Therefore, exactly what a given TPS component expects to be contained in a given input Context, or what is provided in output Contexts of a function is entirely the decision of a given translator. The only information which translators must define for each component is *(1)* What *kind* of component it is and *(2)* What output bindings of variables (if any) a given component produces .

The specific bindings a component outputs gives Planners all the information they need to correctly connect components together and further give Planners the required information to perform certain types of optimisation whilst remaining independant of the specific functionality of the components. The component types (to which a translator can classify a given component) are one of the following:

1. **SourceProvider:** A source provider can instantiate a source of Context instances. This design choice of "potential to construct" rather than simply "an instance of" a source is in direct contrast to the rest of the TPS components. However this extra level of flexibility was seen to be a extremely useful for many types of sources, some of whose instantiation might result in early item consumption from external streams of data.

2. **JoinableComponent:** Joinable components expect to receive the output of Sources (the things made by SourceProviders) as their input and emit bindings of variables. Further, Joinable components expect to be combined together with other Joinables in an AND fashion (see Section 3.4.2). The two subtypes of JoinableComponent instances are: *(1)* Filter patterns and *(2)* Other Translated Production System instances .

3. **Predicate:** Predicates consume a set of bindings and output the same (or potentially augmented) set of bindings. This output is simply the input itself if the predicate is indeed a boolean function. However, in the example of Jena Functors, some predicates may alter input contexts. Once all Joinables are connected to one another, Predicates expect expect to be attached to each other.

4. **Consequences:** Once all Joinables have emitted and Predicatess have been passed, the bindings are handed to these consequences which model what the system expects to do with bindings. For example: a SPARQL SELECT consequences might simply emit certain requested binding variables while a SPARQL CONSTRUCT statement might construct and emit triples from bindings. This construct is subtly different from the Operation described by Planners in Section 4.2.3. Where con-

sequences create the output of a give production system defenition, operations *do something* with these consequences.

5. **TranslatedProductionSystems:** For fully expressing key aspects of most production systems a facility must be provided to allow expression of logical UNION operations or more specifically, forks in the translated production system. By adding a sub TPS to a given root TPS, a Translator is defining a set of clauses that must be combined in an OR fashion (as opposed to the Joinable's AND fashion) with each other. Examples of this are the UNION or OPTIONAL operator in SPARQL.

.

A translator must wrap up the functionality of all parts of a given production rule system as one of these components and specify the variables that particular component creates. The key design choice here is that the bare minimum information is provided to the Planners such that, after the appropriate translation stage, there is a decoupling between the particular production system language and the rest of the production rule system instantiation process.

The general process of a combination of lexer and translator can be seen in Figure 4.1.

### 4.2.3 Planners

Once the translation step is completed, the job of the Planner is to design a Planned Production System (PPS) in the form of a Directed Graph (digraph). The PPS consumes a TPS and plans how its components should communicate in order to answer the question posed by the production system. The digraph defined by the Planner is made up of a set of verteces called NamedNodes and a set of edges called NamedStreams which connect NamedNodes together. As well as providing a guaranteed unique name, a NamedNode instance can express whether it is a[2]:

1. **Source:** The beginning of a graph. The input of the production system is produced here.
2. **Function:** A node in the middle of the graph. The work of the translated production system is done here.
3. **Operation:** A node at the end of the graph. The operation nodes hold the function which must be performed with the final outputs of the production system.

The NamedStreams of a Planner define a vertex-unique named stream, and the bindings which must be grouped on that stream of communication. Though bounded by the bindings which a given compoenent can produce, this list of variables encapsulates the slightly

[2]These 3 states are defined as convenience functions as they could be readily understood by investigating the children and parents of a given node.

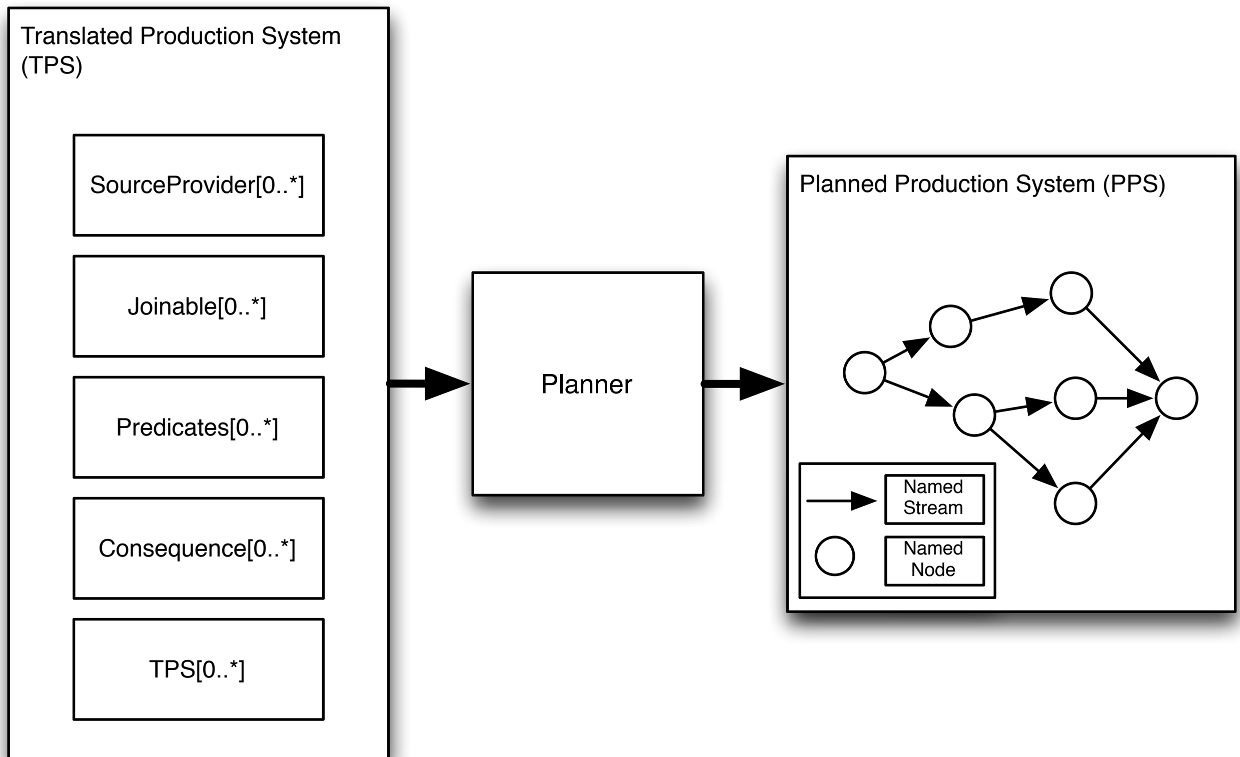


Figure 4.2: Planners consume TPS instances from Translators and produce PPS instances ready for Builders

separate notion of "grouped variables". This concept allows for the potential distribution of Planner nodes. Armed with the knowledge of the grouped variables a distribution-aware Builder can guarantee messages sent across a given stream appear in the correct distributed instance. Planners provide the machinery to join the bindings of the Joinable-Components defined by the Translator, appending in series the Predicates of the translator and also appending the Consequences of the Translator. Exactly how these connections are achieved is a matter for optimization and specific Planners.

The output of all final Consequences are useless unless their use is defined outside the production system. Therefore, along with a TPS, a Planner is given an Operation which is called with all outputs of all final consequences. This operation can be used by specific instantiations of production rule systems to achieve something with the outputs of the system.

At this stage it is important to emphasize what exactly Planners provide and what they disregard entirely. Planners express how data flows through a production system. The Planner's Nodes encapsulate the functionality of the production system. This includes both that functionality defined explicitly by the Translator, but also extra functionality

added by the Planner encapsulating the notion of Translator component joining as well as the final Operation handed to the planner. The Planner's Streams encapsulate the notion of how nodes communicate and what data must appear together with other data when transmitted through this stream. Beyond this, the planner is in no way concerned with the specific functionality of the non-join nodes and the specific transport mechanism of the streams. Further, planners make no attempt define exactly how the outputs of nodes are to be sent to each other, or indeed what they send! The only thing defined is which nodes should receive the outputs of which other nodes. This makes planners both Translator agnostic and Builder agnostic. This separation of concerns is a fundamental design decision of our framework and allows for the decoupled provisioning of arbitrary translators and the realization of production systems through arbitrary Builders.

### 4.2.4 Builders

Finally, builders instantiate and run the digraph defined by the PPS. As should be expected by now, builders make no attempts to understand the functions implemented at each node, nor exactly what each node is attempting to send to every other node. However, armed with the knowledge of the which nodes are connected, exactly how they are connected and exactly what variables must be grouped when items are transmitted through a specific stream, a builder can implement the digraph defined by the planner on arbitrary frameworks processing frameworks.

## 4.3 Example

The framework described in the previous section is perhaps best understood with a concrete example. Here we outline a Jena Translator fed into a Greedy Planner instantiated by an in memory, non-distributed OpenIMAJ[3] stream Builder. Through descriptions and code examples exactly what each level of the framework is responsible for will be better understood.

### 4.3.1 Jena Translator

The Jena Rule language provisions inference rules composed by a body to match which has a head as the consequence of the rule. Each element of the head or body of a Jena rule can be a triple pattern, a functor or another rule. The triple pattern is a triple of nodes where nodes can be literal patterns or URIs, but also variables and wild-cards. Functors are defined by their name and a set of variables and often define predicates over matched variables as well as more complex functionality.

[3] `http://openimaj.org`

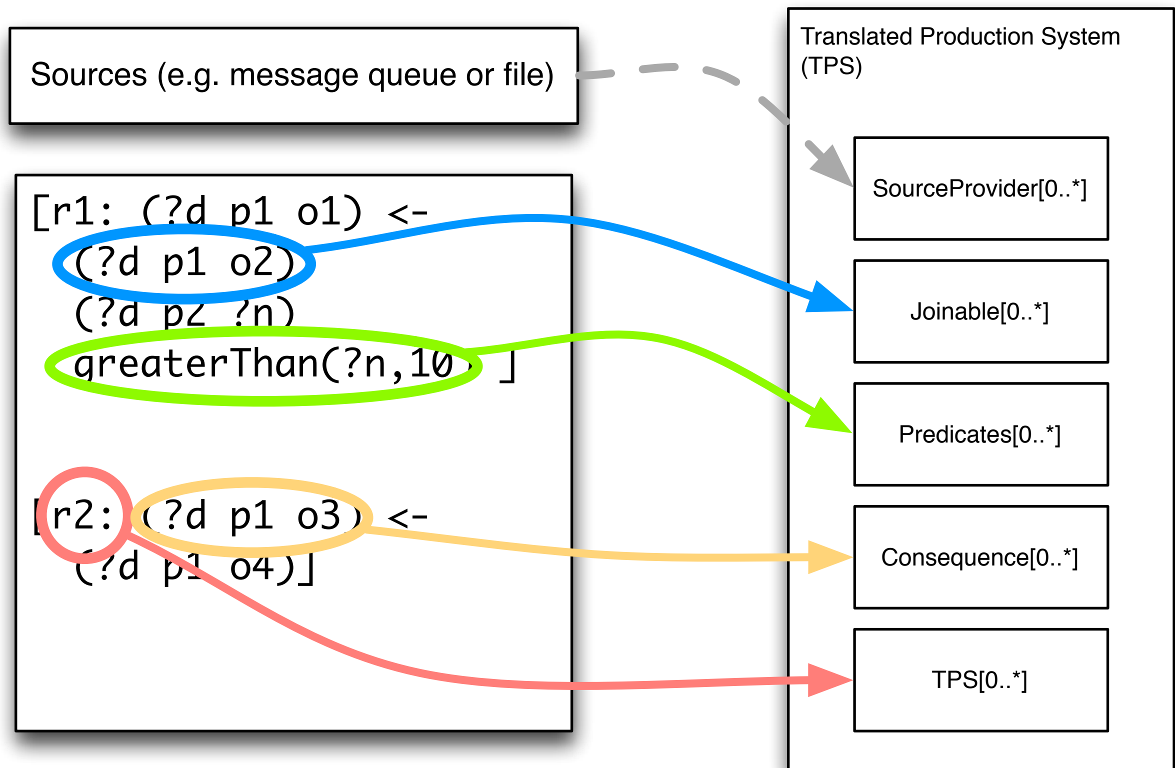


Figure 4.3: Jena translators compile multiple rules as seperate sub TPS of a root TPS. Each sub TPS holds patterns as joinables, functors as predicates and rule heads as consequences. Jena sources are provided to the translator separately.

The Jena translator takes as input a set of rules and an explicit list of input stream sources. Unlike some other production system definition languages which might be supported, Jena rules make no provision for the definition of input streams themselves, so for our production rule system these must be provided. The constructed TPS provisions this source at the root level while each rule in the set of Jena rules is provisioned as a sub TPS.

For an individual rule, the body is handled first. Each component of the body is handled by a specialised function added as a different type of component to the sub compile production rule systems. For each triple pattern, the Jena Translator adds a Joinable component to the compiled production system which consumes a raw triple and emits bindings if any were matched. For each functor, the translator adds a predicate component which runs the functor by extracting variables from the input context and emitting the same or augmented bindings as the output context.

Finally, for the head of the jena rule the translator similarly produces a component for each part of the head, but instead of part of the joinables or the predicates, these components are added as consequences. The consequences which can be constructed are either

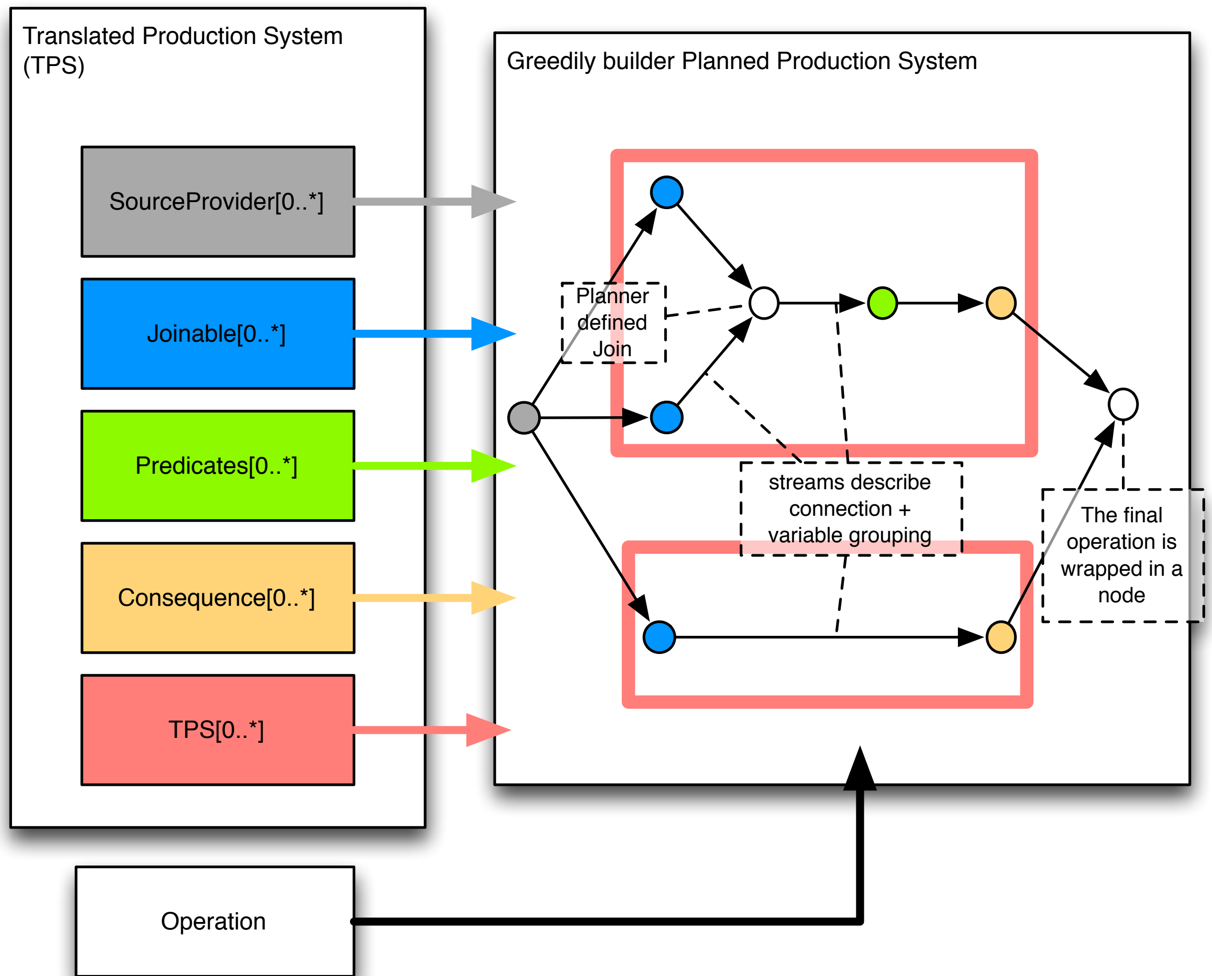


Figure 4.4: The Greedy Planner constructs new paths for sub TPS, Join Nodes between joinables and a final Operation node for the provided Operation.

triple pattern consequence functions (consuming bindings, emitting triples) or functor consequence functions (consuming bindings, emitting nothing).

In Figure 4.3 we show two example Jena rules being compiled into a TPS using this approach.

### 4.3.2 Greedy Planner

As defined in the previous section, the greedy planner is given a Translated Production System (TPS) and a final Operation as input. The Greedy planner makes no attempt to match variables, reuse patterns or optimise join ordering or filters in any way. Instead

the planner creates its digraph in a greedy fashion. The planner starts by creating nodes in the digraph for each source in the TPS. Once done each Joinable in the TPS is connected to all the identified sources of the current TPS. Once connected to their source, each Joinable is greedily (first come, first served) joined with the next Joinable in the order added to the TPS. This resultant "joined combination" is then itself joined with the next Joinable in the TPS. This process proceeds recursively until a single node remains which represents the combination of the Joinables of a given TPS. At this point the predicates defined in the TPS are connected in series to the final Joinable, again resulting in a single node now representing the joinables and the predicates. At this point, sub TPS are treated. Each sub TPS results in a fork in the digraph constructed by the greedy planner, representing the union required. Finally, once all sub TPS of a given TPS are handled, all the resultant nodes are connected to the consequences of the TPS. From the root TPSs, all consequences provided against all the sub TPSs are connected to the final operation which consumes all final consequences and provides the output of the production rule system.

In Figure 4.4 we show the same two example Jena rules after being translated and now being planned as a digraph by a Greedy Planner.

### 4.3.3 OpenIMAJ Stream Builder

OpenIMAJ (Open Intelligent Multimedia Analysis in Java) has recently been extended to support single thread non-distributed stream processing constructs. Stream constructs are defined and augmented using the application of *map* or *filter* operations and are consumed using *foreach* constructs. We choose to demonstrate the idea of Builders in Squall by defining the OpenIMAJ Stream Builder (OIBuilder). The key constructs within the OpenIMAJ stream framework which must be understood so an OIBuilder can be understood are as follows:

**Stream** - At a fundamental level a stream is an object which can provide some "next item" of a given type and can signal when no further items exist. At this level the operations of a stream are fairly close to standard Java iterable objects. Beyond this, streams provide various functional constructs, namely: *map*, *filter*, *transform* and *foreach*. The first 3 operations simply return another stream whose items are that of the original stream with the appropriate modification made as defined by the specific operation. The final operation is the thing which drives the consumption of the stream.

**Map** - When map is called on a stream with a function, each item on the stream is handed to the function and each output is emitted on the new stream. Importantly this is a one to one mapping, and so each object on the original stream has a mapped object on the output stream.

**Filter** - When filter is called on a stream with a predicate, each item on the stream is

checked against the predicate and emitted on the output stream given that the predicate passes.

**ForEach** - When foreach is called on a stream with an operation the consumption of the stream commences. The foreach function the stream it was applied to for the next item available. This might request an item from some parent stream once it has been passed through a map or a filter function. This process of next item request proceeds up the entire stream chain until the original stream is reached. Once a given item passes all maps and filters, the foreach function is given the item. In this way streams are processed and outputs created in a "pull" manner in OpenIMAJ.

**Multiplex Stream** - A multiplex stream wraps around a given stream and allows for multiplex consumption by multiple downstream stream operations. This is achieved by replacing the map, filter, transform and foreach operations with multiplex implementations. These operations create a new "sibling aware" stream object which is aware of all other streams created against the multiplex stream. When any one of these sibling streams consumes from the wrapped stream, a queue is informed on all other siblings. In this way all siblings are given the output of the parent stream in order.

**Join Stream** - A join stream can be considered the opposite of a multiplex stream. It is handed multiple streams on instantiation and presents them all as a single stream. When child streams request an object from this join stream the join stream requests an object from each stream it was instantiated with in a round robin fashion. If any given stream returns an object, that object is returned, otherwise the next stream is interrogated for one round. If no streams return a valid object, the join stream finally returns null. Depending on desired functionality Join Stream instances might be used with buffered non-blocking null-returning maps.

The OIBuilder is handed the digraph generated by any Squall Planner. For each source NameNode a new OpenIMAJ stream is created, wrapped in a Multiplex stream and added to a map of "prepared" streams by the name of the NameNode. All the children of each NameNode is then added to a list of disconnected nodes. This list of disconnected nodes is then investigated such that it is checked whether all the parents of a given disconnected node are prepared and ready to be connected to. Once this is the case for a given node, it is connected to its parent, either using a Join stream (if there are multiple parents), or simply connected directly, in both cases adding the node's function as a map. The map call results in a new stream which is added to the list of prepared streams by name, and again that node's children are added to list of disconnected nodes. This processes proceeds recursively until the length of the disconnected list is equal to 1 and the single item in the disconnected list has no children (an end condition guaranteed by Squall Planners). The operation defined by this final node is connected to all its parents, with a join stream if required, resulting in a constructed and running OpenIMAJ stream instantiation of the production rule system

## 4.4 Conclusion

In this section we have presented our four stage framework for the definition, optimization and instantiation of production systems. When components are written against this framework they can take advantage of other components whilst in development and production, allowing for easy and quick prototyping and support for novel sources of production rules, optimization functions or production system instantiations. In the future we hope to increase the number of example modules created against this design and therefore increase its potential as a useful framework against which to develop.

# Chapter 5

# Conclusions and Future Work

## 5.1 Conclusions

In this work we have outlined and delivered Squall, a novel tool for reasoning and querying against streams of semantic data. In the first instance, we have delivered a command line tool and library which can be used to instantiate local and distributed versions of Rete stream processing networks which can reason against a set of rules and answer queries expressed as CSPARQL.

Though usable, our initial system is explicitly built using certain query languages, constructs query plans optimised against specific algorithms and instantiates these plans against the storm stream reasoning framework. Therefore, we have also delivered an advanced version of the Squall framework which facilitates the instantiation of production rule systems in a more flexible and modular way. The framework delivered should allow for the support of arbitrary languages for: the expression of production rule systems; the reusable and modular expression of production rule system plans; and the ability to physically instantiate these production rule systems against arbitrary stream processing frameworks which may be local, distributed or work in batch settings.

## 5.2 Future Work

The main item of future work arising from this deliverable is an in-depth evaluation of the Squall reasoner, which will be carried out against the semantic data originating from the machine learning deliverables from WP3 and the ontology-driven information extraction tools from WP2. In addition to this evaluation, we have identified the following three areas of future development:

### 5.2.1 Retraction Support

At present, the Squall reasoner produces entailments from input streams which consist of newly asserted facts, rather than a list of assertion and retraction events. In terms of the Stanford STREAM system(Arasu et al., 2003), this corresponds to the output of the IStream relation to stream operator; this approach was taken on the grounds that entailed triples are likely to be as ephemeral as the ground triples from which they were derived. However, if the entailed triples are persisted, this potentially leads to a situation in which there may be conflicting entailments; in this case, it would be appropriate if out-of-date entailments were removed. While this could be carried out in a naive manner by expiring persistent triples based on their timestamps (effectively treating the persisted entailment as a sliding window on the entailment stream), it could also be carried out by introducing the equivalent of STREAM's DStream operator, which produces a stream of newly retracted facts.

### 5.2.2 Ontology Translation

The majority of reasoning tasks on the Semantic Web arise from the use of RDFS and OWL ontologies. While the ontology languages used may have rule-based axiomatisations that can be used to build query plans in Squall (for example, the non-normative axiomatisation of RDFS in (Hayes, 2004) or the OWL2-RL dialect), the ontologies themselves are not inherently rule based and they cannot be easily integrated with Squall (as effectively static data). Moreover, the rule-based axiomatisations of ontology languages typically contain very general patterns which yield Rete-based query plans in which the alpha network does not adequately discriminate between incoming triples; this does not lend itself to the effective distribution of the reasoning process.

However, a rule-based ontology language axiomatisation can be combined with a domain ontology by rewriting each rule as multiple specialised rules, each of which contain some part of the domain ontology. For example, the RDFS entailment rule *rdfs9* ($\{\langle ?u$ rdfs:subClassOf $?x\rangle, \langle ?v$ rdf:type $?u\rangle\} \vdash \langle ?v$ rdf:type $?x\rangle$) can be combined with the triple $\langle B$ rdfs:subClassOf $A\rangle$ to yield a new rule $\langle ?v$ rdf:type $A\rangle \vdash \langle ?v$ rdf:type $B\rangle$. We can therefore convert an ontology defined in a language with a rule-based axiomatisation into an equivalent set of rules that are compatible with Squall.

To this end, we are currently working on a Translator module for ontologies (called `OWLRuleCompiler` in the package `org.openimage.squall.compile.owl` in the main GitHub repository). This module will perform static reasoning over a provided ontology using the Jena libraries, then query the model according to a provided rule set (e.g. RDFS entailments) specified in RIF (W3C's Rule Interchange Format). The provided rule set will be then be populated with the concepts and roles defined in the provided ontology based on the query results returned; This will result in a new set of rules that expresses the ABox reasoning task of the provided ontology, up to the expressivity of the provided rule

set (an OWL 2 Full ontology translated using the RDFS entailments will produce only the rules representing RDFS reasoning over the ontology's ABox). This set of ontology specific rules will be compatible with the behavior of Squall, and could be passed to any Planner module.

# Bibliography

# Bibliography

Anicic, D., Fodor, P., Rudolph, S., and Stojanovic, N. (2011). EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW '11: Proceedings of the 20th international conference on World Wide Web*.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., and Widom, J. (2003). STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26.

Babcock, B., Babu, S., Motwani, R., and Widom, J. (2002). Models and Issues in Data Stream Systems. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems*, pages 1–16.

Barbieri, D. F., Braga, D., Ceri, S., della Valle, E., and Grossniklaus, M. (2009). C-SPARQL: SPARQL for continuous querying. In *WWW '09: Proceedings of the 18th international conference on World wide web*. ACM.

Barbieri, D. F., Braga, D., Ceri, S., della Valle, E., and Grossniklaus, M. (2010). Incremental reasoning on streams and rich background knowledge. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC'10)*, pages 1–15. Springer.

Boley, H. and Kifer, M. (2010). RIF Basic Logic Dialect. W3C Recommendation REC-rif-bld-20100622, World Wide Web Consortium.

Bolles, A., Grawunder, M., and Jacobi, J. (2008). Streaming SPARQL - Extending SPARQL to process data streams. *Proceedings of the 5th European Semantic Web Conference (ESWC2008)*, page 15.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M., Hellerstein, J., Hong, W., Krishnamurthy, S., Madden, S., Reiss, F., and Shah, M. (2003). TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*.

della Valle, E., Ceri, S., Barbieri, D. F., and Braga, D. (2008). A first step towards stream reasoning. In *Proceedings of Future Internet - FIS 2008*, pages 72–81.

della Valle, E., Ceri, S., van Harmelen, F., and Fensel, D. (2009). It's a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Intelligent Systems*, 24(6):83–89.

Forgy, C. (1979). *On the Efficient Implementation of Production Systems*. PhD thesis, Department of Computer Science, Carnegie-Mellon University.

Garcia, J., Montiel-Ponsoda, E., Cimiano, P., Gmez-Prez, A., Buitelaar, P., and McCrae, J. (2012). Challenges for the Multilingual Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 11.

Gromann, D. and Declerck, T. (2012). Terminology harmonization in industry classification standards. In Gornostay, T., editor, *Proceedings of CHAT 2012: The 2nd Workshop on the Creation, Harmonization and Application of Terminology Resources*, pages 19–26. Linkping University Electronic Press.

Gupta, A., Mumick, I. S., and Subrahmanian, V. (1993). Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166.

Hayes, P. (2004). RDF Semantics. W3C Recommendation REC-rdf-mt-20040210, World Wide Web Consortium.

Hoeksema, J. and Kotoulas, S. (2011). High-performance Distributed Stream Reasoning using S4. In *Proceedings of the First International Workshop on Ordering and Reasoning (OrdRing2011)*.

Jin, C., Carbonell, J., and Hayes, P. (2005). ARGUS: Rete+ DBMS= Efficient Persistent Profile Matching on Large-Volume Data Streams. In *Proceedings of the 15th International Symposium on Methodologies for Intelligent Systems (ISMIS2005)*, pages 156–170.

Krieger, H.-U. (2010). A general methodology for equipping ontologies with time. In *7th International Conference on Language Resources and Evaluation*. ELRA. Oral presentation.

Krieger, H.-U. (2012). A temporal extension of the hayes/ter horst entailment rules and an alternative to w3c's n-ary relations. In *7th International Conference on Formal Ontology in Information Systems*. IOS Press.

Krieger, H.-U. (2013). An efficient implementation of equivalence relations in owl via rule and query rewriting. In *Proceedings of the 7th International Conference on Semantic Computing*. IEEE, IEEE.

McCrae, J. and Unger, C. (2014). Design Pattern for Engineering the Ontology-Lexicon Interface. In Buitelaar, P. and Cimiano, P., editors, *Towards the Multilingual Semantic Web*. Springer.

McGuinness, D. L. and van Harmelen, F. (2004). Owl web ontology language overview. Technical report, W3C - World Wide Web Consortium.

Montiel-Ponsoda, E., Vila-Suero, D., Villazn-Terrazas, B., Dunsire, G., Escolano, E., and Gmez-Prez, A. (2011). Style Guidelines for Naming and Labeling Ontologies in the Multilingual Web. In *Proceedings of International Conference on Dublin Core and Metadata Applications 201*, pages 105–115. Dublin Core Metadata Initiative.

Perez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45.

Terry, D., Goldberg, D., Nichols, D., Oki, B., Terry, D., Goldberg, D., Nichols, D., and Oki, B. (1992). Continuous queries over append-only databases. *ACM SIGMOD Record*, 21(2):321–330.

Walavalkar, O., Joshi, A., Finin, T., and Yesha, Y. (2008). Streaming Knowledge Bases. In *Proceedings of the Fourth International Workshop on Scalable Semantic Web Knowledge Base Systems*.

Westerski, A., Iglesias, C. A., and Rico, F. T. (2011). Linked opinions: Describing sentiments on the structured web of data. In *4th international workshop Social Data on the Web (SDoW2011)*, Bonn, Germany.

Wimalasuriya, D. C. and Dou, D. (2010). Ontology-based Information Extraction: an Introduction and a Survey of Current Approaches. *Journal of Information Science*, 36:306–323.