The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Engineering and Computer Science (CECS)

Performant Interoperable Reasoning on a Decentralised Semantic Web

— 24 pt Honours project (S2/S1 2021–2022)

A thesis submitted for the degree
Bachelor of Philosophy (Science) (Honours)

By:
Jesse M. Wright

Supervisors:
Dr. Armin Haller
Dr. Qing Wang

June 2024

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

June, Jesse M. Wright

"Information must be made ***seamlessly available*** *on* ***any device***"

SIR TIM BERNERS-LEE

Acknowledgements

"If I have seen further it is by standing on the shoulders of Giants"

SIR ISAAC NEWTON

In order to develop a complex system that is ready for use in *research* and *production*, we engaged various members of the Semantic Web community. I want to thank all involved for their valuable contributions and insights.

In particular, I would like to acknowledge the following staff members at SolidLab Flanders (University of Ghent):

- Dr Ruben Taelman - the creator of the Comunica Engine, with whom I had several conversations around the high-level architectural *requirements* for implementing reasoning on Comunica. I would also like to thank Ruben for his fast responses (even during his honeymoon) to proposed modifications for the code of the Comunica Engine, which allowed me to quickly incorporate fixes and features required for my project into the core of Comunica.
- Dr Pieter Bonte, who provided his reasoning expertise in conversations around the high-level architecture for implementing reasoning on Comunica.
- Dr Ruben Verborgh, Professor of Decentralised Web Technology at IDLab of Ghent University, Solid Ecosystem Architect for Inrupt, and author of core RDFJS packages such as AsyncIterator, N3.js and LDflex. Ruben has generously given his insights and constructive feedback into the implementations of those packages, which resulted in improved performance and generalisation of the code. Ruben also assisted in clarifying the thesis scope to emphasise *seamless ecosystem integration* rather than only the *scalability* of reasoning.
- Dr Jos De Roo, Semantic Reasoning expert and developer of the *EYE* reasoner, who challenged me to improve the performance of my N3.js implementation on

deep taxonomies.

- Dr Pieter Colpaert, who offered to review my implementation of reasoning in Comunica.
- Dr Patrick Hochstenbach and Dr Joachim Van Herwegen, with whom I had online discussions around the standardisation of rule and reasoning interfaces in RDFJS.

I would also like to thank Mr Jacopo Scazzosi, with whom I collaborated to implement some of my proposed optimisations for the AsyncIterator library. In particular, Jacopo worked on the implementation of the *Wrapping*, *Linked-List* and *Cloning* optimisations.

I would also like to thank my Academic Supervisors at ANU, Dr Armin Haller and Dr Qing Wang. Dr Armin Haller inspired several novel research directions, in particular:

- Restricting reasoning to only inspect select parts of the raw data. This suggestion inspired our future work discussion around the use of ShapeTrees and RDF* to select parts of the RDF dataset to reason over.
- Detecting the reasoning profiles that have been applied to databases by detecting the presence of 'indicator' quads.

Dr Qing Wang provided valuable insights into the structure and presentation of the thesis.

I would also like to thank Dr Sergio José Rodríguez Méndez, who provided outstanding mentorship on my first project in Semantic Web technologies during our work on the Schímatos project only one year into my undergraduate degree.

I would also like to thank my friends and family who have supported me up to, and throughout, my honours year. In particular, I would like to thank Adelaide Grisard. Adelaide, your company allowed me to remain happy, sane and calm during a busy and unpredictable period. Thank you for always supporting me, and I hope that we can continue to be there for one another and grow together.

Finally, I thank my parents for all of the support that they have given me over the years. Without your endless nurture of my curious spirit - I would not be where I am today.

Abstract

The Web is transitioning away from centralised services to a re-emergent decentralised platform. This movement generates demand for infrastructure that hides the complexities of decentralisation so that Web developers can easily create rich applications for the next generation of the internet.

In this thesis we introduce the Typescript based Federated and Interoperable RDF Reasoning Engine (T-FIRRE). Through the development of T-FIRRE we fill a gap in existing research and infrastructure, creating a production-ready reasoning engine for the Resource Description Framework (RDF) that can *interoperably* reason over decentralised documents in a Web client. The architecture of T-FIRRE is capable of applying reasoning algorithms at different layers of abstraction to achieve *high-performance* and *federated* reasoning.

In this thesis, we also demonstrate for the first time that it is *feasible* to perform *fast* rule language (RL) profile reasoning within the browser by developing an *index-based* reasoner for the N3.js library. This reasoner outperforms all previous browser-based reasoners by 10x on *shallow* reasoning tasks and outperforms existing reasoners by *several* orders of magnitude on *deep* reasoning tasks. This reasoner also outperforms state-of-the-art *server-side* reasoners on widely used benchmarks. This additional reasoner also is of benefit to T-FIRRE which, by design, can delegate reasoning tasks to improve performance.

Our T-FIRRE architecture is modular and integrates with the RDFJS compliant Comunica Query Engine, which is now widely used in many Web applications. This integration enables the seamless adoption of our work across a wide range of industry and academic applications.

Table of Contents

List of Figures

List of Tables

Terminology

- **Bindings** A dictionary containing the results of a *SPARQL* query. The keys in the dictionary are *RDF variables* and the values in the dictionary are *RDF terms*.
- **Data view (in RDF)** The set of RDF quads/triples that a system or user can *find*, and has permissions to *retrieve* during a given *query* or *session*.
- **Database** Unless otherwise stated, throughout this thesis, a database is a piece of software or remote service that stores *knowledge graph*s.
- **Data source** Any source of RDF data including *RDF Documents*, *RDF Stores*, *Databases* or a *SPARQL Endpoint*.
- **Enrichment** Enhancing queries or data-sets by adding *implicit data*, making the results more 'rich' in data.
- **Explicit Data** Data that has been explicitly placed in a database, either by human input or by being imported from other data sources.
- **Fact** Synonymous with a *triple*, unless explicitly stated otherwise.
- **Federated Operation (Reasoning/Querying)** The practice of executing an operation over multiple local *and/or* remote documents *and/or* databases, and producing results that are the same as those that would be returned if the operation was executed over the union of all the data sources.
- **Implicit Data** Data that is output by an *automated reasoner* which takes explicit facts and rules as input.
- **Inference (abbr. Infer)** The process of generating *implicit data* using an *automated reasoner*.
- **Internationalised Resource Identifier (IRI)** A superset of URIs which uses the Universal Character Set rather than being limited to ASCII.
- **Interoperability** The ability for different systems to work with one-another. *Semantic Interoperability* occurs when these systems are able to *accurately* interpret the information that is exchanged between them.

- **Knowledge Graph (KG)** A *set* of *triples* or *quads*.
- **Link-Traversal-based Query Processing** Link-Traversal-based Query Processing (LTQP) Hartig (2013a) is a research area exploring techniques to query over a linked set of Linked Data documents, by following the links between them. Existing tools for LTQP include LDSpider Isele et al. (2010) and SQUIN Hartig (2013b). LTQP is currently being researched within the Comunica Query Engine[1].
- **Materialisation** The process of generating *implicit data* and placing it into a *knowledge graph*.
- **Ontology** Within computer science, an ontology (Gruber, 1993) is used to provide *meaning* to data. Specifically, they provide formal *names* and *definitions* for categories, properties and relationships - which are used to describe *entities*.
- **Ontology Language** An ontology language provides a formal definition for the *logic* and *syntax* of an ontology (Corcho and Gómez-Pérez, 2000).
- **Promise** A JavaScript object that 'represents the eventual completion (or failure) of an asynchronous operation and its resulting value'[2]. A promise may be in one of three states: *pending* in which it is neither fulfilled or rejected; *fulfilled* which means a given operation was completed successfully; or *rejected* which means the operation failed.
- **Quad** An RDF triple, with an additional field denoting the *named graph* that the triple belongs to.
- **Query Rewriting** The process of modifying a *SPARQL* query such that when executed over only *explicit data* the results are the same as if the original *SPARQL* query had been executed against the *implicit* and *explicit* data.
- **RDF Store** An in-memory object used to store *knowledge graphs*. We consider an RDF Store to be a type of *Database*.
- **RDF Document** A local file or remote Web page containing RDF triples or quads.
- **RDFJS** The RDF JavaScript Libraries Community Group[3] sets standards for the APIs of RDF terms, data sets, and query engines in JavaScript. They also publish a set of machine-readable and enforceable TypeScript[4] declarations[5] to ensure implementors comply with the standards. This enable greater interoperability between various JavaScript libraries for RDF.

[1] https://comunica.dev/research/link_traversal/
[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[3] https://www.w3.org/community/rdfjs/
[4] https://www.typescriptlang.org/
[5] https://github.com/rdfjs/types

- **Resource Description Framework (RDF)** A standardised model used for data interchange, particularly on the Web.
- **Rule** A declarative machine-readable specification that is used to define how an *automated reasoner* should generate *implicit data* from a set of *explicit data*.
- **Software Development Kit (SDK)** A set of software and tools that can be used to build applications.
- **Semantic RDF Reasoner** Software that infers *implicit data* typically using *explicit data* and *rules*.
- **Semantic Web** A set of Web standards designed to make the Web interoperable and machine-readable.
- **SOLID** A specification designed to enable individuals to securely store their data in decentralised data Pods. Users can control which individuals, organisations and applications can access this data.
- **SPARQL Protocol and RDF Query Language (SPARQL)** A declarative query language to query over RDF data.
- **Triple** An atomic RDF statement about an RDF resource. A triple consists of a *subject*, *predicate* and *object* in a tuple. For instance, the statement that *Tim Berners-Lee is a computer scientist* is made with the triple `(timbl:me, a, ex:computerScientist)`.
- **Universal Resource Identifier (URI)** A unique sequence of characters used to identify a *resource* (logical or physical) used by Web technologies.
- **Uniform Resource Locator (URL)** The subset of URIs that indicate the *location* of resources (such as on the Internet, a private network, intranet or filesystem).
- **Web** The abstract 'space of information' consisting of documents, media and information connected by hypertext links.

Part I

Prologue

Chapter 1

Introduction

Modern Web infrastructure is predominantly built in a centralised manner, hosted by a small number of influential organisations such as Meta, Amazon, Netflix and Google (De Filippi and McCarthy, 2012). In contrast, Tim Berners-Lee's original vision for the Web was to create a democratic, distributed platform on which 'anyone can say anything about anything'[1]. Globally, there is a growing interest[2] (EUd) in data privacy and sovereignty on the Web, largely driven by problems around data exploitation[3] (Chaston et al., 2015) and censorship[4] (Stjernfelt and Lauritzen, 2020) that arise within the centralised model. Consequently, the Web ecosystem is moving back towards a decentralised model, as seen through the growing uptake of Linked Data (Bizer et al., 2008) and the SOLID protocol (Capadisli et al., 2021; Mansour et al., 2016). These protocols build on the Semantic Web (Berners-Lee et al., 2001b) technology stack.

Accordingly, researchers are increasingly seeking to develop ways for applications to interface with a decentralised Semantic Web, which addresses a key hurdle to its adoption. Front-end developers need tooling that is as *accessible* (Verborgh and Taelman, 2020), *intuitive* and *efficient* to use as traditional APIs for accessing centralised resources[5] (Pautasso and Wilde, 2009). Until now, decentralisation research has focused on interoperable discovery and querying of Semantic Web data. There existed a fundamental gap in the research of the enrichment of decentralised data on the Semantic Web, through the addition of implicit facts into the set of query results.

In typical centralised databases containing real-world Semantic Web data, implicit facts comprise a sizeable portion of the database. For instance, in the UniProt (Consortium,

[1] https://www.w3.org/DesignIssues/Metadata.html
[2] https://gdpr-info.eu/
[3] https://www.theguardian.com/technology/live/2018/apr/10/mark-zuckerberg-testimony-live-congress-facebook-cambridge-analytica
[4] https://quillette.com/2019/06/06/against-big-tech-viewpoint-discrimination/
[5] https://ruben.verborgh.org/blog/2021/12/23/reflections-of-knowledge/

2015) knowledge graph of protein sequences, 46.1% of the 228 million facts within the database are implicit. Further, in the Classical Art Research Online Service (CLAROS) (Kurtz et al., 2009) knowledge graph, 81.4% of the 102 million facts are implicit. Considering only ABox reasoning on ontologies such as OpenCyc[6], an ontology of general human knowledge, the effect is even more pronounced; 99.8% of the 1176 million ABox facts in the database are generated by rules (Motik et al.). Therefore, effectively performing reasoning on a decentralised Semantic Web is necessary to producing equally rich results.

Given the importance of implicit data to centralised Semantic Web applications, there is an immense opportunity for decentralised applications to have similar reasoning functionality. To enable this, client-side Resource Description Framework (RDF) reasoning is an emerging area of research (Terdjimi et al., 2015), which has the capacity to unlock a wide range of IoT applications; ranging from user-empowered social media (Werbrouck et al., 2019) to live health diagnoses (Sondes et al., 2019). However, the limited work on client-side RDF reasoning (Terdjimi et al., 2015) has failed to produce a reasoner that meets the *performance*, *interoperability* and *usability* requirements for many of these applications.

1.1 Decentralisation in the Semantic Web

The Semantic Web facilitates a decentralised 'serverless' development environment in which agents and applications can connect to different data sources depending on use-case requirements. This is attractive for several reasons:

1. Privacy: Rather than trusting centralised service providers such as Google and Facebook, users can choose to collect and share data with 'trusted insiders', such as friends, organisations and services (Olmedilla, 2007). For instance, the SOLID project (built on top of Semantic Web protocols) provides users with fine-grained control over their data use and access. Additionally, cryptographic burdens-of-proof in data sharing are being developed to more effectively trace and prosecute privacy breaches (Diallo, 2018; Gholami and Laure, 2016; Blanchette, 2012).

2. Scalability: Web services increasingly need to handle requests from a large number of clients with complex tasks, necessitating scalable solutions to prevent a slow-down. Decentralised versions of the Semantic Web offer a solution, by shifting the computational burden onto the client for complex tasks such as query evaluation. For example, Triple-Pattern-Fragment servers allow clients to execute a SPARQL query by first retrieving *fragments* of data that match a particular quad pattern, then performing the computations for a result (Verborgh et al., 2016; Terdjimi et al., 2015). Another emerging example is FOG (or edge) computing architectures (Yi et al., 2015) which distribute reasoning and query execution across

[6] https://github.com/asanchez75/opencyc

several IoT devices, which may improve the *efficiency* and *security* of data transfer, reasoning and querying (Le et al., 2020).

3. Flexibility/Mobility: Not all IoT scenarios necessarily include a set-up with readily available server-side technologies; for instance in internet black spots, or locations with low-bandwidth/high latency. Yet, individuals may still wish to retrieve, reason and query over data that is available to them; for instance, data that is published by local sensors that are connected to a mobile phone via Bluetooth. In these cases, it is imperative to provide a platform for easy development (Terdjimi et al., 2015). However, in all of the aforementioned applications, it is not just the explicit data that we are interested in, we are also interested in implicit data that can be obtained from the various data sources across the Web of things. Moreover, we are interested in making this data available to developers in such a way that it looks and feels as though the data was explicit in the first place so that we can enrich IoT applications without introducing any development cost/overhead.

1.2 Problem Statement

RDF reasoning does not occur in browser-based applications, despite being critical to the next generation of Web technologies. The primary constraints are that client-side reasoners do not meet the *performance* requirements of modern applications and the technicalities of using reasoners make them *inaccessible* to front-end developers. Additionally, there is minimal research into *interoperable* RDF reasoning on the Semantic Web, despite the the importance of universal interoperability to the success of decentralisation projects such as the SOLID project.

Several factors lead to these deficiencies. Most existing research on Semantic Web reasoning has targeted centralised, or server-side applications as discussed in Chapter 3. In conjunction, it is only now becoming possible to execute reasoning efficiently in the browser as a result of the recent performance improvements in JavaScript execution engines[7].

Accordingly, our research objectives are:

- **R01 Interoperable Architecture** Develop a modular architecture that enables *interoperable* and *federated* RDF reasoning. This architecture should enable one to enforce assumptions about the rules, and explicit facts that are used to generate implicit data included in query results. We shall call this architecture T-FIRRE.
- **R02 Performance** Demonstrate that it is *possible* to perform reasoning in the browser in a manner that is *performant enough* for *standard use-cases* to execute without interrupting the user experience in applications that make use of the reasoner. In addition we should investigate what reasoning performance is *truly possible* within the browser.

[7] https://blog.chromium.org/2021/05/chrome-is-faster-in-m91.html

- **R03 Accessibility** Create an architecture (T-FIRRE) for reasoning that is easy for Web developers and researchers to use. As part of this, the architecture should comply with RDFJS standards so that any developments can be easily integrated into existing RDFJS applications and libraries.
- **R04 Future-Proof Design** The T-FIRRE architecture that we develop should foster future work by being *extensible*. In particular our work should enable new federated reasoning algorithms and applications to be easily developed and tested. Additionally, it should be simple to experiment with reasoning in orthogonal research directions such as link-traversal.

In summary, the challenge is to implement an *interoperable* and *performant* client-side reasoner that is *accessible* for researchers and front-end developers to use now and into the future.

1.3 Contributions

Our work makes a significant contribution toward enabling interoperable RDF reasoning, introducing T-FIRRE: the first *flexible* and *performant* reasoning architecture to be run in a browser. This is particularly important for emergent extensions of the Semantic Web, including Linked Data and SOLID Protocols.

In this work, we firstly develop an interoperable reasoning *architecture* for T-FIRRE that is capable of applying RDF reasoning at several abstraction layers. This architecture is designed so that it is guaranteed that reasoning is applied to data *prior to* the execution of queries, enabling developers to enforce assumptions around the inference that has been performed on their data. Simultaneously, this architecture pushes reasoning to the lowest possible abstraction layer that is available, so as to enable *performant* RDF reasoning. Finally, T-FIRRE is compliant with RDFJS standards, allowing our work to be integrated into existing applications and libraries off-the-shelf.

We further develop the theory and implementation required to enable *interoperable*, *performant* and *accessible* RDF reasoning in browser clients. This thesis makes two key contributions in order to achieve this. We firstly develop RDF reasoners in JavaScript which have competitive performance when benchmarked against state-of-the-art server-side reasoners. This enables a *seamless* end-user experience in apps that rely on the reasoner internally. Secondly, we develop our reasoner using an architecture that enables front-end developers to seamlessly *use*, *customise* and *extend* the tool.

T-FIRRE consists of production-ready reasoning components that are compatible with the Comunica Query Engine (Taelman et al., 2018). Comunica is a widely adopted SPARQL query engine used in *web browsers* and *web servers*. These components of T-FIRRE use novel algorithms (and data-structure optimisations) we have developed to enable *performant* and *interoperable* federated reasoning over remote data sources. Furthermore, we have written more than 150 unit tests for these T-FIRRE components

to obtain 100% test coverage in the software packages we have released.

1.4 Thesis Structure

This thesis aims to prove that *performant*, *interoperable* and *federated* client-side RDF reasoning is possible. In Part I we motivate why this work is required by decentralisation projects that are built on top of the Semantic Web. We then achieve this goal with the development of T-FIRRE in Part II. T-FIRRE brings reasoning capabilities to the browser and enables off-the-shelf query enrichment through integration with the Comunica Query Engine. We then demonstrate the exceptional reasoning performance that is *truly possible* in JavaScript, by implementing reasoning directly over the indexes of the N3.js RDF store in Chapter 11. This work on N3.js is integrated into T-FIRRE as T-FIRREs architecture can enable reasoning at the lowest possible abstraction layer for a given set of sources. We conclude our thesis in Part III with a *quantitative* evaluation of our goals, before discussing directions for future work.

Chapter 2

Theoretical Background

This chapter introduces the key technical concepts required to understand the contributions of this thesis. We first outline the area of automated computational reasoning and then introduce the Semantic Web, where the research is applied. Following this, we will examine how these areas intersect to give the domain of our research, RDF reasoning; and outline the fundamental concepts within this domain.

2.1 Automated Reasoning

Automated Reasoning is a large area in Computer Science that is integral to many fields of research including automated theorem proving and machine learning.

Broadly speaking - reasoning is defined as "the ability to make inferences [about data]" (Portoraro, 2001) and automated reasoning is "concerned with the building of computing systems that automate this process".

The term is often associated with deductive reasoning in formal logic/mathematics. Throughout this thesis, we are interested in the more 'constructive' practice of deducing (inferring) *implicit* (inferred) facts (information) from explicit data.

Whilst we are yet to formalise this idea, we can build an intuition for it through an example. Suppose we are told that *Tim Berners-Lee is a computer scientist* and *computer scientists are human*. Then we can deduce the *implicit* fact that *Tim Berners-Lee is a human*.

Whilst such information seems obvious, there is a need to programmatically deduce such information and databases - and the true power of such systems is seen when many layers of implicit facts can be used to produce results that are not obvious to the human eye.

2.1.1 Horn Logic

Throughout our work, we are primarily interested in working with reasoning via rules that satisfy the constraints of Horn Clauses. A Horn Clause (Horn, 1951) is a clause (logical disjunction) with at most one positive literal (i.e. unnegated atom). Such clauses can be written as a logical implication with a conjunction of (unnegated) atoms in the premise, and a single atom as a conclusion.

For instance, the statement:

$$\text{if } `x \text{ is a } y' \textit{ and } `y \text{ is a subset of } z' \text{ then } `x \text{ is a } z'$$

may be considered a valid Horn Clause - since the premise is a logical conjunction and the conclusion is a single atom. However,

$$\text{if } `x \text{ is a } y' \textit{ or } `y \text{ is a subset of } z' \text{ then } `x \text{ is a } z'$$

is not a valid Horn Clause as it contains a *disjunctive* premise, and

$$\text{if } `x \text{ is a computer scientist' then } `x \text{ is a Human' and } `x \text{ is a Fish'}$$

is also not a valid Horn Clause as it contains a conjunction in the conclusion.

2.1.2 Declarative Programming Languages - Datalog and Prolog

Many automated reasoners are implemented as declarative *Datalog* or *Prolog* reasoners. For instance, RDFox (Nenov et al., 2015), a state-of-the-art reasoner on many RDF reasoning benchmarks, is a Datalog reasoner.

Prolog. Prolog (Clocksin and Mellish, 2003) first appeared in 1972 as a logic programming language (Apt, 1990) based on formal logic. Prolog is designed as a largely *declarative* (Lloyd, 1994) language, where the program logic is written in terms of facts (for instance `professor(timbl)` or `knows(jesse, timbl)`) and rules, such as `professor(X) :- person(X)`. A computation in Prolog is instantiated by running a query (for instance `?- person(timbl)`) and the query is subsequently evaluated by the Prolog interpreter.

All rules and facts within Prolog are *Horn Clauses* making it the ideal candidate for RL profile RDF reasoners, including the EYE (Verborgh and De Roo, 2015) proof engine for Notation3 logic.

Datalog. Datalog Huang et al. (2011) first appeared in 1986 as a weakly typed subset of Prolog. Datalog is *completely* declarative, and Datalog queries made on finite sets are guaranteed to terminate. Since Datalog is a subset of Prolog, all programs are still formed from sets of facts and rules, and computations are performed by executing a query over a program. Datalog is widely regarded as *the* query language for deductive databases.

2.1.3 Implementation/Execution of Prolog/Datalog

Logically most implementations of Prolog / Datalog use a resolution (Robinson, 1965) method called SLD (Gallier, 2015) (Selective Linear Definite) clause resolution. SLD resolution typically proceeds by attempting to refute the negation of the query. If the negation can indeed be refuted, then the query is found to be true.

2.2 The Semantic Web

The Semantic Web[1] (Berners-Lee et al., 2001a) is a technology stack developed by the W3C[2] to enable interoperability between systems. At its core is the concept of Linked Data, which is the 'collection of interrelated [and machine readable] datasets on the Web'[3]. Linked Data is achieved by publishing and exchanging data in the standardised Resource Description Framework (RDF) (Wood et al.).

The atom of a piece of data in RDF is called a triple. A triple relates a resource (a *subject*) to another resource or value (an *object*) through a predicate (a *verb*). A knowledge graph is a set of triples that link and describe such resources and entities, and the triple can be thought of as an edge in the graph. It is common to include multiple named graphs within a database[4], triples in named graphs are represented as quad - where the fourth value in the tuple is the identifier for the named graph. Vocabularies (or ontologies)[5] are used to define concepts and relationships (predicates) in RDF datasets.

Sitting on top of RDF and Linked Data are a couple of key concepts, which are summarised in the Semantic Web Layer Cake (Berners-Lee, 2002) in Figure 2.1. The first key concept is that of Query[6]; SPARQL Protocol and RDF Query Language (SPARQL) queries are used to execute queries on the Semantic Web. There are several ways that this may be achieved: first, if linked data endpoints expose their data via such an endpoint; and second, by embedding a SPARQL query engine within client applications so that they can retrieve data and execute queries over it.

Another pillar on top of RDF and linked data is the idea of inference[7]. Inference is the use of rules and explicit facts, to 'enrich' a dataset with implicit data. Inference can also be used to detect inconsistencies within datasets.

[1] https://www.w3.org/standards/semanticweb/
[2] https://www.w3.org/
[3] https://www.w3.org/standards/semanticweb/data
[4] https://www.w3.org/TR/n-quads/
[5] https://www.w3.org/standards/semanticweb/ontology
[6] https://www.w3.org/standards/semanticweb/query
[7] https://www.w3.org/standards/semanticweb/inference

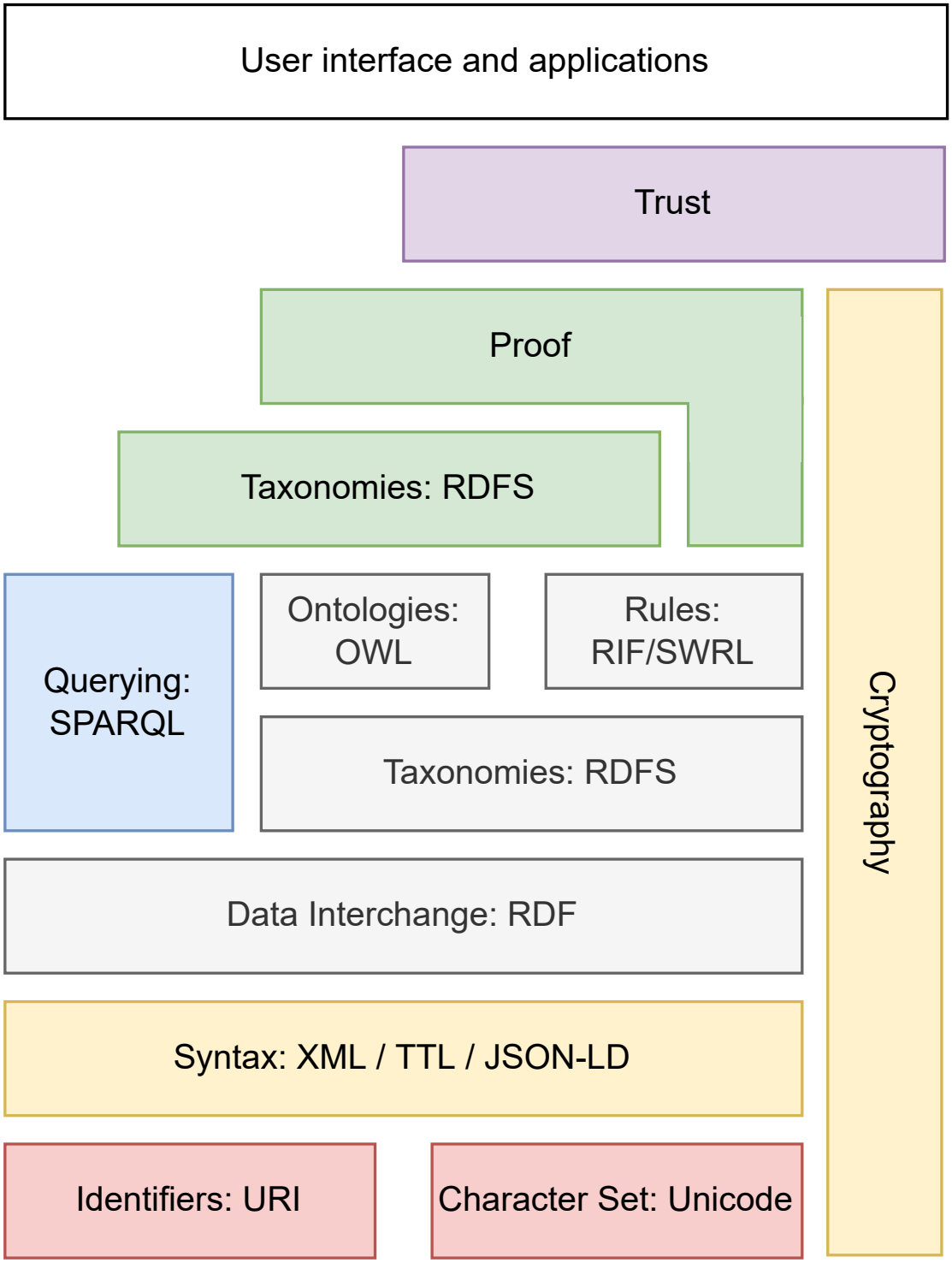

Figure 2.1: The Semantic Web Layer Cake - where each layer uses capabilities from the layers below it (Berners-Lee, 2002).

2.3 Triples, Quads & Quad Patterns

The concept of Quad Patterns underlies a large portion of our work and as such, it is imperative to have a clear understanding of their functionality.

2.3.1 Triples & Quads

Data is typically represented as a triple within RDF. For instance, the statement that *Tim Berners-Lee is a computer scientist* would be described using the triple

`(timbl:me, a, ex:computer_scientist)`.

A *knowledge graph* then can be formed from a *set* of such triples.

It is often desirable to have distinct *graphs* of information within the same database. These graphs, for instance, may represent data from different users or points in time. This is solved through *Named Graphs* Carroll et al. (2005), which allow users to specify the graph that data belongs to according to a given URI. If a triple does not sit within a *named graph*, then it is considered to sit within a *default graph*.

In order to be able to capture the information of which *graph* a triple belongs to, the concept of a *quad* is introduced. Quite simply, it contains an extra entry that denotes which graph the triple belongs to. For instance, if the information that *Tim Berners-Lee is a computer scientist* exists within the graph `ex:my_graph` then this would be captured using the *quad*

`(timbl:me, a, ex:computer_scientist, ex:my_graph)`

If the data exists within the default graph, then the data is instead encoded within the quad

`(timbl:me, a, ex:computer_scientist, DEFAULT_GRAPH)`

2.3.2 Triple Patterns & Quad Patterns

A *quad (or triple) pattern* is a *quad (or triple)* where any of the entities in the tuple MAY be left undefined. For instance

`(timbl:me, a, ex:computer_scientist, DEFAULT_GRAPH)`,

`(timbl:me, a, _, DEFAULT_GRAPH)` and `(timbl:me, _, _, _)`

are all considered to be *quad patterns*. *Quad patterns* are typically used to reference, or request, the set of data in a document or database that *matches* the given pattern.

2.3.3 The Ontology Web Language (OWL2)

The Ontology Web Language (OWL2) (Motik et al., 2009) is a *structural specification* that provides a language used for expressing ontologies that is used in the Semantic

Web. OWL 2 ontologies are used to define the classes, the properties, the individuals, and the data values that are used within Semantic Web applications. OWL2 ontologies are themselves, stored as Semantic Web documents. The syntax and semantics of the language provide a foundation on which APIs and reasoners can be implemented.

Within OWL2 there are two common semantics: *Direct Semantics*[8] and *RDF-Based Semantics*[9].

Direct Semantics in OWL2 are based on the SROIQ flavour of *description logic*, whilst *RDF-Based Semantics*[10] considers a restricted set of semantics, in which rules are expressible as a Horn Clause.

The abstract reasoning interfaces that we implement in this work are compatible with both forms of semantics. In a restricted browser environment, most concrete discussions and implementation of reasoners will focus on *RDF-Based Semantics* out of a desire to reduce the complexity of reasoning algorithms.

2.4 RDF Reasoning

Throughout this project, we are interested in rules-based semantic reasoning on RDF data sets so as to infer logical consequences from an asserted set of facts (axioms).

There are two primary techniques used in literature (Mishra and Kumar, 2011; Rattanasawad et al., 2013) to achieve the process of semantic reasoning, which are known as forward chaining and backward chaining.

2.4.1 Forward Chaining

Forward Chaining (Finin et al., 1989) begins with a data source and recursively applies rules until new results are achieved. This is generally considered an eager method for fact materialisation, as all facts must be realised before the database can be queried.

Forward chaining is often called materialisation as systems materialise (pre-compute) facts and place them in memory. The most straightforward technique for forward chaining is the naive fixed-point algorithm (Broekstra and Kampman, 2003) which repeatedly re-applies rules to all facts in the database until no new facts are produced. The semi-naive algorithm (Ceri et al., 1989) improves upon this by only evaluating rules with at least one premise containing an implicit fact generated from the previous round of reasoning.

However, there are some optimisations that can be made to these algorithms in contexts where the query to be performed by an engine is known in advance. An example of this

[8] https://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/
[9] https://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/
[10] https://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/

is implemented within the closed source materialisation algorithms of RDFox (Nenov et al., 2015; Motik et al., 2014).

As (a small) part of our work, we implement and extend this algorithm.

2.4.2 Backward Chaining

Backward chaining (Russell and Norvig, 2009, p. 337) first begins with a set of goals (facts) - which it wishes to test to see if they are true. The inferencing engine then examines the consequents of rules in the rule-set. For any consequents that match the goals we wish to test, the engine then checks to see if the antecedent is supported. This can happen one of two ways:

1. Data supporting the antecedent is present within the dataset.
2. The consequent of another rule supports the antecedent - and the process is repeated (chained) to see if the antecedent of the downstream rule is supported.

This backward chaining approach is akin to the SLD approach typically seen in Prolog and Datalog systems as discussed in Section 2.1.3.

2.4.3 Incremental Reasoning

Incremental reasoning (Reyes-Alvarez et al., 2014; Maarala et al., 2016) is used to maintain up-to-date reasoned views on databases, as new data is inserted and deleted.

A major argument in support of the development of incremental reasoning is that it will lead to capabilities for *stream reasoning* Stuckenschmidt et al. (2010) which is necessary in the new age of constantly changing data. Similarly, another interesting use-case for incremental reasoning is in link traversal (Hartig, 2013a), where new sources are being detected *throughout* the query and reasoning process. In this, use-case views of implicit facts must be maintained as additional views of the data become available.

Incremental reasoning algorithms calculate changes to implicit data sets in response to updates in explicit data. The goal is to improve the performance of reasoners that reason over constantly changing data sets, and avoid the need to completely re-derive a fact set.

The principle of incremental view maintenance was first introduced by Gupta Gupta et al. (1993) who presented the Delete and Rederive (DRed) algorithm for SQL databases.

Using the set of *explicit* data that is to be deleted, the algorithm first deletes a superset of the implicit triples that need to be removed. In turn, the algorithm recreates the data that *did not* need to be deleted and adds any new implicit facts that are to be added based on the new *explicit* facts that are added.

The HyLAR engine (Terdjimi et al., 2015) adapted and implemented a version of this DRed (Delete and Rederive) algorithm for RDF reasoning. The technique it follows is

first to restrict its reasoning ruleset to only those rules with a premise matching those quads which are in the set of staged deletions. It repeats this with any new triples that are to be deleted.

Once this superset of implicit facts has been removed, the rederivation cycle begins.

The Backward/Forward Algorithm (Motik et al., 2015) is a more intelligent incremental reasoning algorithm that supports incremental updates of Datalog materialisations. This allows RDFox to outperform most prior reasoners because 'existing solutions, such as the well-known Delete/Rederive (DRed) algorithm, can be inefficient in cases when facts have many alternate derivations [as such facts will constantly be deleted and then rederived as the data changes]' (Motik et al., 2015).

2.4.4 Description Logic

Description Logics (DL) (Baader et al., 2003) are a subset of First-Order Logic (FOL) that were developed to describe knowledge. Specifically, they are designed to model and reason over the relationships between concepts, roles and individuals. Description Logics provide the formalisms for the OWL2 ontology that is core to the Semantic Web.

Description logics are less expressive than FOL, with the benefit of being decidable. Description logics are also more expressive than propositional logic.

The Attributive Concept Language with Complements ($\mathcal{ALC}$) is widely regarded as the *primitive* description logic within most Semantic Web applications. More expressive Description Logics are then created for these applications by adding further semantics and syntax to $\mathcal{ALC}$.

Terminological Box - TBox. TBox axioms are used to define *concepts* and the relationships thereof. For instance, the statement *all computer scientists are human* is a subset relationship between the concepts of *computer scientist* and *human*.

Assertional Box - ABox. ABox axioms on the other hand describe the relations between entities (for instance *Tim knows John*) and how entities relate to concepts (for instance *Tim is a computer scientist*.

(Concept-Based) Knowledge Base. A knowledge base is defined as a tuple $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ with $\mathcal{T}$ a set of TBox and $\mathcal{A}$ a set of ABox axioms. The benefit of this distinction is that it allows in theory, to enable each of the two components to be reasoned over in isolation (De Giacomo and Lenzerini, 1994). In turn, this can improve reasoning efficiency. However, in practice, this distinction does not often exist in many Semantic Web applications, especially in a decentralised context. Consequently, for the majority of this thesis, there will be no distinction made between TBox and ABox axioms when reasoning is performed.

Open World Assumption - OWA. The Open World Assumption states that if information is *not known to be truthful* within a system, it *is possible* that it still holds true (Keet, 2013). The consequence of this is that the *inability* to prove a fact with a reasoner *does not* imply that the negation of that fact holds true.

The Attributive Language with Complements - $\mathcal{ALC}$. As previously mentioned, most Description Logics used within the Semantic Web contain the syntax and semantics of $\mathcal{ALC}$. The $\mathcal{ALC}$ logic contains the fundamental notions of atomic negation, intersection of concepts, universal restrictions, existential quantifiers, and concept negation of concept names (Harald Karlsen, 2015a,b; Cerami, 2014; De Rijke et al., 1998).

2.4.5 OWL2 Reasoning Profiles

Within OWL2 there are several formally defined *reasoning profiles* (Sengupta and Hitzler, 2014), which are designed to trade off the *computational complexity* and *expressivity* of rules within rule-sets. The standard profiles are given in Table 2.1.

2.5 Linked Data Fragments

There are three traditional ways of accessing linked data:

1. A *data dump* of all triples in the *entire* dataset.
2. A *subject page* with all data about a specific subject.
3. A *SPARQL result* from a SPARQL Construct query[11].

Linked data fragments (LDFs) are used to describe the superset of such views. LDFs have a method of selecting data (for instance by IRI, SPARQL Query or Quad Pattern) and return metadata in addition to raw data. This metadata includes variable names, result counts, and controls for the service (such as links to related pages).

The design of a particular LDF protocol trades-off query expressivity for improved sever availability and load capacity.

2.5.1 Triple Pattern Fragments

Triple pattern fragments (TPFs) are a standard that formalises a particular type of linked data fragment. The core idea of TPFs is that data queried using Triple Patterns has a *low evaluation cost*. Secondly, these TPFs are paginated so that results can be retrieved 'on-demand' by clients, and contain metadata such as the estimated total number of results so that clients are able to develop intelligent querying strategies.

[11] https://www.w3.org/TR/rdf-sparql-query/#construct

Table 2.1: Overview of common OWL2 reasoning profiles

Profile	Complexity Class	Overview	Use Cases
OWL 2 EL	PTIME-complete	Provides constructors to perform subsumption and instance checks	Applications with large amounts of hierarchical information such as species data
OWL 2 QL	NLogSpace-complete	Provides constructors for rules that can be applied through query rewriting to provide sound and complete conjunctive query answering	Enables use cases where implementors wish to reason over existing relational databases using query re-writing,
OWL 2 RL	PTIME-complete	Allows rule-based reasoning with Horn Clauses	Applications that require scalable reasoning, whilst maintaining moderate expressivity - such as Web-based reasoning
OWL 2 DL	Decidability open	Based on description logic (specifically $\mathcal{SRIOQ}(\mathcal{D})$ Tsarkov and Horrocks (2006)), and is the most expressive reasoning profile that is *also* decidable, sound and complete	Commonly used in Web applications
OWL 2 Full	Decidability open	Contains all OWL 2 DL and RDF(S) capabilities. This profile is not guaranteed to terminate the reasoning process.	Applicable to all RDF documents

2.5.2 Quad Pattern Fragments

Quad Pattern Fragments extend (and are backward compatible with) the TPF specification so as to add support for Named Graphs (Carroll et al., 2005).

Chapter 3

Existing RDF Reasoning Implementations

RDF reasoning has existed since the inception of the Semantic Web in 2001 (Berners-Lee et al., 2001a). Since then, the majority of RDF reasoner implementations have been developed in languages such as Java or C++ and target desktop or server environments (Khamparia and Pandey, 2017). Moreover, these implementations are generally tightly coupled to a particular database implementation and hence cannot perform federated reasoning against multiple data sources.

In general, existing RDF reasoners focus on the task of scalable reasoning for large data sets stored in a single Knowledge Graph on a server. As an example, DBPedia (Auer et al., 2007) is a large RDF knowledge graph that was created as a community effort to extract structured data from Wikipedia. By 2014, the Virtuoso (Erling and Mikhailov, 2009) backed knowledge graph contained more than 3 billion RDF triples[1]. DBPedia also provides RDF links to other datasets where possible to improve interoperability with other datasets on the web. Another such graph is Wikidata (Vrandečić and Krötzsch, 2014), which is a crowd-sourced knowledge graph founded in 2012, that follows Wikipedia's model of open-editing. At the time of writing, Wikidata contains more than 13 billion triples.

Server-side enrichment, however, is insufficient for achieving an *interoperable* and *decentralised* version of the Semantic Web such as SOLID. Within such an ecosystem everyone has access to different views of explicit (and in turn implicit) RDF data, and this data may be spread across multiple locations such as local files, SOLID Pods[2] and public knowledge-graphs. Here, *client-side* reasoners are required to 'fill in the gaps' when extending the inferences to consider multiple sources *or* adding reasoning to documents

[1] http://wikidata.dbpedia.org/about
[2] https://solidproject.org/users/get-a-pod

delivered locally or by less-intelligent servers.

Consequently, developments in such server-side reasoning technologies (Wood et al.; Tsarkov and Horrocks, 2006; Glimm et al., 2014) do not compete with the objectives of this thesis, but rather complement them. In particular, the interoperable architecture we propose allows us to make use of pre-materialised *globally viewable data* from public *knowledge graphs* such as DBPedia and Wikidata. In turn this improves reasoning performance by eliminating the need to re-calculate inferences. Concurrently our interoperable design can supply additional inferences based on data that is only visible to certain users.

Furthermore, there has been limited research into performing RDF reasoning across federated data-sources (Sakr et al., 2018), despite there being extensive research into federated query processing (Acosta et al., 2019; Taelman et al., 2018; Verborgh et al., 2016).

3.1 RDFox

RDFox (Nenov et al., 2015) is a commercial solution for RDF storage, reasoning and query evaluation that is largely considered state-of-the art with respect to benchmarks on scalability and performance. RDFox is a centralised RDF store which uses main-memory storage; and performs reasoning using forward-chaining in parallel Datalog programs. RDFox supports backward and forward chaining reasoning strategies. The reasoner also supports incremental reasoning via the novel Backward/Forward algorithm that the authors introduce (Motik et al., 2015).

3.2 EYE Reasoner

The EYE Reasoner (Verborgh and De Roo, 2015) is a Prolog reasoner compatible with NodeJS[3]. It performs RDF reasoning and answer querying via forward, and backward chaining. However, it does not yet support reasoning in-browser due to blockers in upstream packages[4].

3.3 HyLAR Reasoner

The HyLAR Reasoner (Terdjimi et al., 2015) performs 'Hybrid Location-Agnostic Reasoning' and was developed to address the lack of JavaScript client-side reasoners. The authors claim the reasoner to have a 'lightweight, modular and adaptive architecture'. The engine they developed supports reasoning via forward-chaining materialisation; and also has support for incremental reasoning through a novel tag-based approach (Terdjimi et al., 2018).

[3] https://nodejs.org/en/
[4] https://github.com/josd/eye/issues/24

However, this reasoner has not seen widespread usage; with only two dependent packages[5] at the time of writing. Some of the drawbacks that are causing this may include:

- **Package Size**: The originally published package for this software is 27.8MB[6], which is considered bulky in the context of browser applications.

  We note our work to extract the core logic of this reasoner resulted in a 92.1KB package[7]. However, this is still much larger than what is required to implement more performant reasoners that are presented later in this thesis.

- **Performance**: Reasoning with the HyLAR engine takes a non-negligible amount of time, even on relatively small data sets. For instance, it takes more than a second to reason over the union of *Tim Berners-Lee*'s profile card and the FOAF ontology when run on an 8GB mobile device.

  This is an infeasible time-scale for many user facing applications which require data retrieval and evaluation to occur in the order of milliseconds in order for a seamless user experience to occur (Nah, 2004).

  This issue with performance is largely because the implementation used a forward chaining technique in which all implicit facts were required to be materialised before the execution of a SPARQL query. This forward chaining technique was also sub-optimal, in that each rule evaluation required iterating over all of the facts in the dataset, rather than using an index to lookup data. Consequently, reasoning on relatively small datasets can have impractically large time and memory requirements.

- **RDFJS Compliance**: The reasoner was written prior to the completion of the RDFJS data model specification[8] and has not since been updated to fully comply with the specification. Consequently, the reasoner requires custom-written functions[9] to convert between RDFJS data-structures and HyLAR facts in order to perform reasoning. In turn this makes it difficult to use with other JavaScript libraries and applications.

- **Lack of Support for Remote Sources**: Despite the authors citing developments in remote data sources, such as Triple Pattern Fragments (Verborgh et al., 2016) in the motivation for their work, users of the HyLAR reasoner are required to separately handle content negotiation, data fetching and data loading, which introduces overhead for developers.

[5] https://www.npmjs.com/browse/depended/hylar
[6] https://www.npmjs.com/package/hylar
[7] https://www.npmjs.com/package/hylar-core
[8] https://rdf.js.org/data-model-spec/
[9] https://github.com/jeswr/hylar-core/blob/95a0dd2ad1ba452512c8fef7afce4c8933bbee5c/lib/ParsingInterface.ts#L26-L64

3.4 Research Gaps

Overall, examining the existing RDF reasoning implementations reveals sizeable research gaps, which present opportunities for immediate high-impact applications. The objectives of this thesis directly respond to these research gaps and exceed existing benchmarks, in particular:

- **R01 Interoperable Architecture** There are currently no designs that enable interoperable and federated reasoning many distinct types of sources. In particular, there is no reasoner that allows users to input a *set* of data-sources, and rules and then obtain the results of federated reasoning. In response, the thesis develops a modular T-FIRRE architecture that enables *interoperable* and *federated* RDF reasoning. This architecture enables one to enforce assumptions about the rules, and explicit facts that are used to generate implicit data included in query results.
- **R02 Performance** There are currently no architectures that enable performant RDF reasoning from within the browser. Our research demonstrates that it is *possible* to perform reasoning in the browser in a manner that is *performant enough* for *standard use-cases* to execute without interrupting the user experience in applications that make use of the reasoner. In addition we should investigate what reasoning performance is *truly possible* within the browser.
- **R03 Accessibility** There are no RDFJS compatible reasoners, so it is difficult to add reasoning capabilities to existing RDFJS libraries and applications. Our research creates an architecture for reasoning that is easy for Web developers and researchers to use. Our research architecture complies with RDFJS standards so that any developments can be easily integrated into existing RDFJS applications and libraries.
- **R04 Future-Proof Design** Beyond the scope of existing work, our architecture aims to foster future work by being *extensible*. In particular our work enables new federated reasoning algorithms and applications to be easily developed and tested. Additionally, it will simple to experiment with reasoning in orthogonal research directions such as link-traversal.

Chapter 4

RDF Reasoning Use-Cases

In this chapter, we explain the need for reasoning on the Web by analysing some common use cases for RDF reasoning. Where relevant, we also outline the need for interoperable and federated Web-based reasoning in these use cases. In Part II, our work will outline optimisations that enable such use-cases to be achieved in a more performant manner.

4.1 Enrichment

The most commonly known and straightforward application of RDF reasoning is to enrich data using RL profile rules and *ABox* axioms (ontological data) to materialise *TBox* data (instance data). The following sections illustrate this.

4.1.1 Subclass Hierarchies

Within most datasets, one does not explicitly state all of the classes that an entity belongs to. For instance, *Tim Berners-Lee* may state within an RDF document that he is a *Computer Scientist*, and make no other assertions about the *class*es that he belongs to. As a result, if the user were to run a query over the database, requesting all of the members of the *human* class then *Tim Berners-Lee* would not be included in the results.

Reasoning over Subclass Hierarchies can be used to improve the results of this query, by using the *ontological* information that *Computer Scientists* are a subset of *Human* and consequently infer the fact that *Tim Berners-Lee* is in fact a human.

4.1.2 Sub-property Hierarchies

Similar to how *subclass* hierarchies can be used to deduce the classes that an entity belongs to, *subProperty* hierarchies can enrich queries over more general *relationships*. For example, if we have the fact that `ex:Alice ex:friendsWith ex:Bob` and we wish to

run a query to find all of the people that *Alice* knows, then reasoning can be performed with the ontological fact that `ex:friendsWith rdfs:subClassOf foaf:knows` to be able to *infer* that `ex:Alice foaf:knows ex:Bob` and hence have `ex:Bob` included in the set of results.

4.1.3 Symmetric Relationships

Another common task for reasoners is to complete symmetric relationships (otherwise known as inverse properties). A common example of this is the concept of *knowing* a person, that is to say; if *Alice knows Bob* then it must also be the case that *Bob knows Alice*. Within a reasoner the rule

`(?p1 owl:inverseOf ?p2) ∧ (?x ?p1 ?y) -> (?y ?p2 ?x)`

can be applied to use the fact that `foaf:knows owl:inverseOf foaf:knows` and `ex:Alice foaf:knows ex:Bob` to generate the result `ex:Bob foaf:knows ex:Alice`.

4.1.4 Class inference Domain and Range

Information can also be used to infer the classes that an entity belongs to based on *other facts* that may be stated about them.

For instance, the ontological statement `foaf:knows rdfs:domain foaf:Person` states that all entities in the *domain* of the `foaf:knows` predicate belong to the class person. This is expressed via the rule

`(?a rdfs:domain ?x) ∧ (?u ?a ?y) -> (?u rdf:type ?x)`

Hence if the fact `ex:Alice foaf:knows ex:Bob` exists within a database that has the FOAF ontology loaded, in addition to the above RDFS rule `ex:Alice a foaf:Person`.

4.2 Schema Alignment

Data on the web is inherently messy - and one of the primary causes of this within *RDF* is as a result of users applying different *ontologies* to represent similar information. For instance, there are many OWL2 ontologies that introduce the concept of an *individual* or *person*. They include the FOAF[1], vCard[2] and schema.org[3] ontologies.

There are many reasons for which this overlap in schemas may occur, including:

- Different use cases requiring different levels of *precision* with which they define concepts.
- The existence of ontologies, such as the *Wikidata ontology* which are *automatically generated* from *instance data* Suchanek et al. (2011).

[1] http://xmlns.com/foaf/spec/
[2] https://www.w3.org/TR/vcard-rdf/
[3] https://schema.org/

- Ontology designers failing to find existing ontologies for the concepts they wish to use.

This is a topic of particular interest[4] within the SOLID community and Flanders Government at present. This includes the *active development* of *rules* and *metadata* to align ontologies that are often used within the SOLID ecosystem[5].

4.3 Entity Resolution

A similar (though simpler) problem to that of Schema Alignment is that Entity Resolution Benbernou et al. (2017) (or *Entity Alignment*) Christophides et al. (2015). This is required when multiple databases may refer *conceptually* to the *same entity* but use different *identifiers* for them. For instance, there are many identifiers on the internet used to reference *Tim Berners-Lee* including:

- https://timbl.inrupt.net/profile/card#me - the identifier used in his personal profile document
- https://www.wikidata.org/wiki/Q80 - the identifier used to reference him in Wikidata[6]
- http://dbpedia.org/resource/Tim_Berners-Lee - the identifier used to reference him on DBpedia[7]

Entity Alignment can be used to *consolidate* these identifiers, along with any statements that are made about them. This is typically done by stating that they are the same identifier using the `owl:sameAs` relation and then applying rules that resolve entities that are semantically the *sameAs* one another.

[4] https://github.com/SolidLabResearch/Challenges/issues/15#issuecomment-1054510245
[5] https://paul.ti.rw.fau.de/ ec69etyl/2022/dkg-22/DKG-22_paper_7.pdf
[6] https://www.wikidata.org/
[7] http://dbpedia.org/

Chapter 5

Interoperable Reasoning Use-Case: Developing the SOLID Ecosystem

In this chapter, we contextualise our work in terms of its importance to the success of the SOLID protocol[1]. The SOLID protocol is an extension of the Semantic Web designed to re-decentralise the Web and give users control over their data. SOLID[2] is a specification that extends existing web protocols to support a more decentralised and *social* web.

The SOLID protocol is relevant to our research because interoperable access to implicit and explicit data is crucial to its success. Moreover, front-end developers are key to enabling lay users to access the SOLID ecosystem. These developers need to be able to easily *access* reasoning capabilities, and these reasoning capabilities need to be *performant* enough so as to avoid slowing down the applications they build. Moreover, since the SOLID protocol is still in its early stages, the standards around it are constantly evolving, and thus a reasoner within the ecosystem must be *future-proof* amid constant change.

At the core of the SOLID specification is the concept of a SOLID Pod in which a user (or a trusted third-party provider) hosts their own data. Individuals can choose to collect and share data with 'trusted insiders', such as friends, organisations and services (Capadisli et al., 2021). Users can also revoke access at any point in time.

5.1 The Need for a Developer Ecosystem

One of the primary hurdles in developing applications for SOLID is the overhead that developers face in handling flows for accessing, enriching, validating, querying and modifying data (Verborgh and Taelman, 2020). Developers must concurrently handle other

[1] https://solidproject.org/
[2] https://solidproject.org/about

workflows such as those around authentication (Coburn et al., 2022). Additionally, these standards have not yet reached a stable status and as such, are constantly evolving, including the introduction of breaking changes.

Consequently, there is a significant overhead to application development. Once applications are developed, they will regularly break as a result of specification changes or be incompatible with certain SOLID server[3] implementations, having made some false assumptions about the specification.

5.2 Existing Work on SOLID

There has been some work on the creation of developer SDKs for SOLID. This includes the Inrupt SDK[4], which contains libraries for authentication[5] and basic API access to SOLID POD data[6]. However, such libraries do not contain functionalities such as data-validation and reasoning. Consequently, app developers are still required to handle these parts of the development process, while simultaneously not having any knowledge of what has already been handled on the server side.

5.3 Developer Ecosystem Overview

The architecture and ideas presented throughout this thesis form part of a larger developer ecosystem that the author, and other members of the SOLID community are working to create. It is the author's view that a key milestone in this ecosystem is the creation of an SDK for front-end developers that enables them to *easily* interact with distributed remote servers that form the back-end of their applications. These developers *should* be able to access, and modify the data, as if it is a *strictly-typed* in-memory object containing *clean* and *consistent* data.

At the same time, the complex tasks of content-negotiation, authentication, query processing and reasoning need to take place under the hood in an *efficient*, *secure* and *stable* manner, so that users can receive 'instantaneous' results when using applications.

In order for this to be possible, there are several pieces of key infrastructure that are required:

1. Authenticated Data Pods containing raw user data.
2. Pod Aggregators containing aggregations of *publicly viewable* data from pods. These include aggregators for single servers, and aggregators that aggregate data over multiple servers. Furthermore, these aggregators should apply reasoning to their views.

[3] https://solidproject.org/users/get-a-pod
[4] https://inrupt.com/products/dev-tools/
[5] https://github.com/inrupt/solid-client-authn-js
[6] https://github.com/inrupt/solid-client-js

3. A Client Side Query Engine that can:
    a) Detect the sources that need to be queried over.
    b) Apply appropriate reasoning over the selected *view* of data, by 'filling in the gaps' on the views that pods, aggregators, and other data sources provide.
    c) Execute a query over those sources, including negotiation with servers to distribute the task between server and client depending on resource availability.
4. Query engine 'wrappers' that hide the complexity of engine configuration, query building, and query execution.

The author identified this need when developing a SHACL (Knublauch and Kontokostas, 2017) form generator (Wright et al., 2020b) for RDF data-entry; having experienced a lot longer period of time on the creation of underlying infrastructure for data retrieval, data-cleaning and data querying, than the core product. In subsequent works - the author developed parts of this stack, including a strictly typed data interface (Wright et al., 2020a) and ML-based server-side reasoners[7]. In the open-source space we have also worked on query engines[8] and query engine wrappers[9].

In this work we continue towards this overarching milestone. In particular, we address the complex challenge of applying reasoning within a client-side query engine in such a way as to fit into the existing ecosystem, whilst simultaneously having a negligible impact on the performance.

[7] https://demo.hedgedoc.org/viJNcZZTTv-GnpwVjLCIbg#Future-Work
[8] https://github.com/comunica/comunica/commits?author=jeswr
[9] https://github.com/LDflex/LDflex/commits?author=jeswr

Part II

T-FIRRE: An Interoperable and Performant RDF Reasoning Engine for the Comunica Engine

Chapter 6

Background on the Comunica Engine

This chapter introduces the Comunica Query Engine and motivates why we have chosen a design for T-FIRRE that enables it to integrate with the Comunica framework. The primary motivation for this decision is that the architecture of Comunica aligns with our objective **RO1 Interoperable Architecture** due to its modular and interoperable design for federated SPARQL queries. The Comunica engine also has competitive performance on many query operations. By using similar data-structures for RDF transfer and storage to that of Comunica, we facilitate our objective **R02 Performance** in T-FIRRE. In Chapter 10 we shall discuss how we further enhanced these internal data structures used by Comunica and T-FIRRE, in order to further improve the performance of the respective engines. Integration with the Comunica architecture also enables **R03 Accessibility** due to the RDFJS compliance of the Comunica engine. Finally, **R04 Future-Proof Design** is achieved by allowing new features to be bridged between the T-FIRRE and the Comunica engine via dependency injection.

6.1 Architectural Background

This thesis is primarily architectural. Hence, we provide background information on the design principles and patterns used throughout this work. In T-FIRRE and Comunica, more generally, four key design patterns are used to achieve a modular design that is adaptable to different run-time conditions.

6.1.1 The Actor Model

T-FIRRE and Comunica make use of the *actor* model to break down computations into different tasks. The actor is a computational 'unit' that performs a specific type of task. The actor is instantiated by receiving a message (input), and then it will act upon that message. In the course of completing its task, an actor can delegate parts of its task by

sending messages to other actors.

The actor model is traditionally used in *concurrent programming* (Hewitt et al., 1973), where an individual actor acts on a single thread and has its own *private state*. Communication with other threads would take place via *message passing* with other actors.

The *Actor Model* is useful within T-FIRRE and Comunica as it:

1. Aids in making the engine configurable and interoperable by allowing the same task to be implemented in different ways by different actors. For instance in T-FIRRE we implement actors for both *abstract* source-agnostic reasoning algorithms and for source-specific reasoning such as reasoning over N3.js stores.

2. Aids in making the engine extensible as new functionality for the engine can be defined by adding new actors that execute new tasks for the engine.

6.1.2 The Publish-Subscribe Pattern

Comunica makes use of the *Publish-Subscribe* pattern to pass *messages* between *actors*. The actor sending the message is known as the publisher and the recipient is the *subscriber*. Instead of the *publisher* and *subscriber* being programmatically coupled to one another, the *publisher* will publish its message to a bus that handles a *category* of messages. Subscribers then subscribe to one (or more) *bus*' and will receive messages that are published to the *bus*.

The primary benefit is that the *publishers* and *subscribers* do not require *a priori* knowledge of one another. Thus, different configurations of *publishers* and *subscribers* can be built to give the engine different properties and capabilities.

In T-FIRRE and Comunica, the *Publish-Subscribe* pattern enables the same task to be implemented in different ways, with each implementation corresponding to a different *actor*. For example:

1. Different implementations of *dereferencing* allow *raw data* to be retrieved from different locations (such as stored on disk, in an in-memory store, or in a remote file).

2. Different implementations of parsing to allow for parsing of different RDF serialisations.

3. Different join algorithms can be selected based on network constraints, the number of quads matching given patterns, or overlap in entities on joined variables.

In T-FIRRE, this *Publish-Subscribe* pattern is particularly useful by enabling:

1. Different implementations of *parsing* to allow for different rule serialisations.

2. Different implementations of *reasoning* to handle different reasoning profiles.

3. Different configurations of *reasoning* to enable various combinations of *lazy*, *eager* and *incremental reasoning*.
4. Different implementations of *rule optimisation*.
5. Different sets of rules to be packaged with the engine.

6.1.3 The Mediator Pattern

The *Mediator Pattern* is used within T-FIRRE and Comunica to further decouple *actors*. Rather than having the *actors* pass messages directly between one another, a *mediator* component passes messages between them. When an actor tries to delegate a *task*, that task will be given to a *mediator*. The *mediator* then mediates that task over the actors that are *subscribed* to the *bus* for the category of the task. Mediators may perform diverse functions, including:

1. Choosing the best (such as fastest or least memory-intensive) actor for the task.
2. Taking the *union* of the responses from all of the actors.
3. *Pipelining* the task sequentially through all of the actors that are subscribed to the *bus*.

6.1.4 Dependency Injection and Components.js

T-FIRRE is not a *single* reasoning engine and Comunica is not a *single* query engine. Rather, each is a collection of components that can be configured in different ways to create engines with different properties/behaviour. This build-time customisation is achieved through *dependency injection* using the Components.js (Taelman et al., 2022) dependency injection framework.

The Components.js dependency injection framework functions by semantically defining modules using the Object-Oriented Components ontology (Van Herwegen et al., 2017). Configuration files are similarly defined using Object-Oriented Components ontology, and Component.js will then build the software from these semantic definitions.

6.2 The Architecture of Comunica

Comunica is 'a modular SPARQL Query Engine for the Web' (Taelman et al., 2018). Comunica consists of a set of packages, which are either an *actor* that performs a specific task, a *bus* that defines the interface for a specific set of tasks, or a *mediator*, which selects the best *actor* from a *bus* to perform a specific task. The interaction between the *actor*s, *mediator*s and *bus*es is shown in Figure 6.1. T-FIRRE uses a similar *actor-mediator-bus* model in its' design. Taelman (2022) shows a default configuration of these *actors*, *mediators* and *bus*es that is used to produce a client-side SPARQL query engine, capable of querying over in-memory, on-disk and remote data sources.

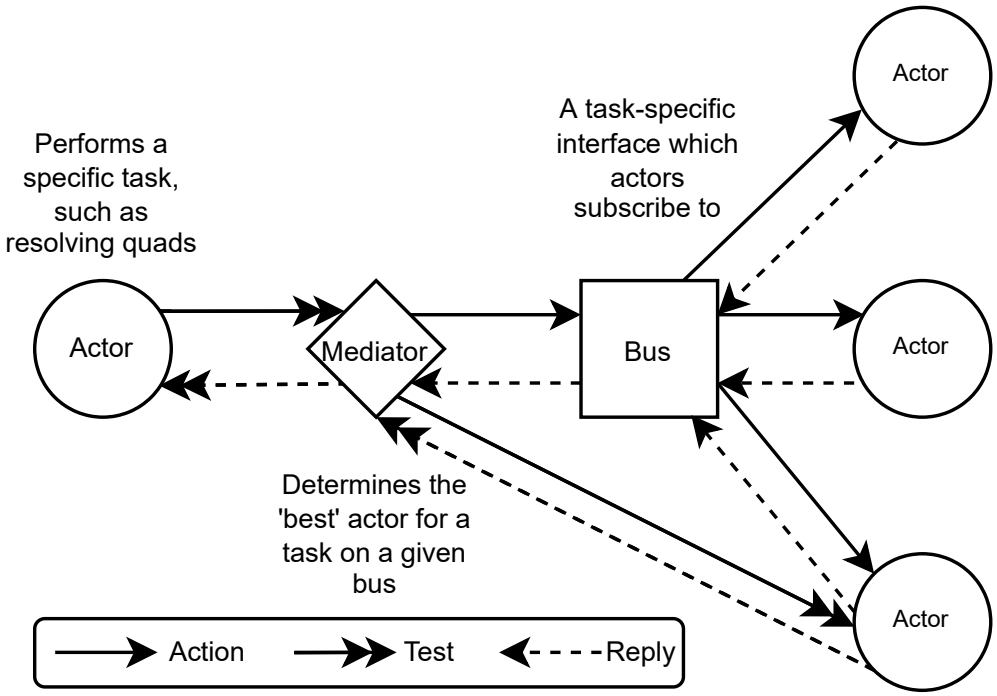

Figure 6.1: The actor-mediator-bus model is used by Comunica. Based on the diagram found in Taelman et al. (2018).

6.2.1 Query Evaluation in Comunica

In order to understand our work in subsequent sections, it is vital to effectively understand how queries are *evaluated* in Comunica - and as such, understand how implicit data is added to queries.

For every operator and expression in the SPARQL query algebra[1], Comunica implements an *actor*. For every query operation actor, except the *quad pattern* operator, the expression will be broken into sub-queries which then get *mediated* to other actors within the engine. These actors will then evaluate the sub-query and return *a pull-based stream* of results. The actor that requested these results will then manipulate these streams to perform its operation. For instance, the `UNION` operator will take a union of all the streams it receives, whilst a `BGP` operator will perform a *join* operation on the stream of bindings that it receives.

When the *quad pattern* query operation actor gets called, the engine will then retrieve all data that matches the given *quad pattern* from each of its' sources.

6.2.2 Resolving Data in Comunica

For our work, it is also essential to understand how *quad patterns* are resolved into asynchronous *pull-based* data streams. The *bus* that handles this action is known as the *resolve-quad-pattern* bus. It accepts a quad pattern as input and returns an *AsyncIterable* quad stream. Several actors implement this *bus* interface - one for each type of data source that Comunica handles and a *federated* actor that is responsible for aggregating results when multiple sources are provided. One source actor is a Quad-Pattern-Fragment (Verborgh et al., 2016) actor, which retrieves quad patterns from a Quad Pattern Fragment server on demand. Other source actors include a SPARQL actor, which retrieves quad patterns from a remote SPARQL endpoint - and an RDFJS actor, which retrieves quad patterns from in-memory RDFJS data-stores using the in-built `match` method.

The federated actor operates by *delegating* the quad pattern matching to actors that handle resolving quads for each specific data source. The federated actor then returns the union of the results from each source.

6.3 The Case for Integration with Comunica

The architecture of Comunica provides an ideal basis for researching and developing reasoning capabilities for IoT devices for several reasons.

Firstly, Comunica already supports *federated* SPARQL queries over and is *interoperable* (**R01 Interoperable Architecture**) with a variety of linked-data sources, including,

[1] https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html

but not limited to, SPARQL endpoints, linked-data fragments, and various serialisations for static RDF Documents.

Additionally, Comunica is *accessible* to developers (**R03 Accessibility**), and has a rapidly growing user base of both *app developers* that wish to create apps for the Semantic Web, or SOLID Project[2]; and *researchers* who wish to develop new algorithms for queries and joins (Brumm, 2019). Furthermore, this is a fully open-source project, with infrastructure in place to ensure long-term maintenance through the Comunica Association[3]. Importantly, Comunica is also compliant with RDFJS standards[4], which define standardised interfaces for representing RDF data, stores and query engines in JavaScript. This makes it easy to integrate with existing libraries and applications.

Another key reason for choosing to integrate T-FIRRE with the Comunica framework to bring reasoning to the client-side of the Semantic Web is Comunicas modular design. This design enables researchers to easily add and benchmark *new* research ideas that they may have by adding modular components that research these ideas, supporting **R04 Future-Proof Design**. A typical use case for this within Comunica has been to experiment with different types of join algorithms for different environments (depending on various factors such as the location of data sources, memory size, processing speed and network latency).

We use this capability in our work to implement and benchmark new proposals for reasoning on the Semantic Web.

[2] https://solidproject.org/
[3] https://comunica.dev/association/
[4] https://github.com/rdfjs/types

Chapter 7

Extending Functionality of the Core Comunica Engine

Throughout our work, we discovered several limitations within the Comunica engine that posed barriers to the development of T-FIRRE and thus achieving our research objectives. This section outlines how we extended the functionality of the Comunica Engine in order to prepare for the development of T-FIRRE.

We consider the work that we have completed and outlined in this section to be fulfilling the pre-requisites required to enable interoperable and federated reasoning within Comunica via T-FIRRE (**R01 Interoperable Architecture**).

7.1 Semantics of Blank Nodes

A *blank node* is a node within an RDF graph used to represent a resource that has not been assigned a unique identifier (URI). Semantically, blank nodes are 'existential variables'.

Due to this lack of universal identification, blank nodes are always *scoped* to an individual document or store (Wood et al.).

Blank nodes are often a pitfall in implementations of RDF, often being misunderstood or even disregarded entirely within applications (Mallea et al., 2011).

7.1.1 Skolemisation

In some circumstances, stronger identification for blank nodes is needed - for instance, when data is aggregated from multiple documents or stores. In these circumstances, systems can replace blank nodes with a new and globally unique Skolem IRIs (Wood et al.).

Skolemisation is used to handle federation within Comunica. Whenever more than one source is available to the query engine, it will prepend the locally scoped blank node identifier with a prefix unique to the source. Consequently, Comunica can maintain blank node semantics for *queries*.

However, prior to our work, Comunica had incorrect behaviour on blank nodes for *updates* which involved blank nodes. For instance consider the update in Listing 7.1.

```
PREFIX foaf: <http://xmlns.com/foaf/spec/>

INSERT {
    ?s a foaf:Agent
} WHERE {
    ?s a foaf:Person
}
```

Listing 7.1: Sample SPARQL Insert Query

executed on several documents, one of which contains the data in Listing 7.2

```
@prefix foaf: <http://xmlns.com/foaf/spec/> .

_:b1 a foaf:Person .
```

Listing 7.2: Sample Data

by the skolemisation process, the blank node `_:b1` in the document obtains the universal identifier `urn:comunica_skolem:source_1:b1` for processing throughout the engine. Since the engine does not perform any form of deskolemisation when skolemised URIs are placed back into the source, the result was that the updated document would look like that in Listing 7.3

```
@prefix foaf: <http://xmlns.com/foaf/spec/> .

_:b1 a foaf:Person .
<urn:comunica_skolem:source_1:b1> a foaf:Agent .
```

Listing 7.3: Actual Result

when the expected result would be like that in Listing 7.4.

```
@prefix foaf: <http://xmlns.com/foaf/spec/> .

_:b1 a foaf:Person, foaf:Agent .
```

Listing 7.4: Expected Result

In order to handle this, we added components into the core of the Comunica Engine that *deskolemise* Skolem URIs when they are being inserted back into the document they were retrieved from.

7.1.2 Skolemisation and Reasoning

In *materialisation* based reasoning, which T-FIRRE implements, reasoners need a location in which to store implicit facts. Within the context that Comunica operates, data sources (such as Web documents) are typically read-only - this means that we cannot add implicit facts to these documents. Moreover, some implicit facts are generated from explicit facts arising from multiple sources, in which case there is no well-defined place in which to place the new fact. To overcome this, we allow users of the Comunica to define a new *implicit destination* in which to place implicit facts. This destination can be any RDF destination that Comunica supports for updates, including SPARQL Endpoints, local files and in-memory stores.

If this destination is treated as a regular RDF document, the semantics of blank nodes will break down when adding implicit facts about blank nodes that appear in a different source. This is because RDF semantics mean that the blank nodes within each document represent disjoint entities. To overcome this problem, we introduce the abstract notion of a **documentExtender**[1]. These **documentExtender**s may be any typical destination for RDF data - but are assumed to *only contain inferences from other documents*. Thus any blank nodes contained in the document are assumed to be Skolem URIs from *other* documents.

A **documentExtender** *should not* be treated with the same semantics as an RDF document of its own. This is due to the aforementioned semantics of blank nodes - and exists purely due to the *technical constraints* of being unable to place data into *remote* sources during the reasoning process. This concept of a **documentExtender** is similar to the idea of blank node scoping introduced in QuadStore[2].

7.2 Dereferencing and Parsing Data

Dereferencing is the process of retrieving and resolving a document from a given URI, that is 'When someone looks up a URI, provide useful information'[3].

On the other hand, parsing is the process of analysing such a document into logical components. In the case of parsing RDF documents, the result of parsing is a set of *triples* or *quads*.

In the original design of the Comunica Engine, the logic for dereferencing data was tightly coupled with the logic for parsing documents. This logic is shown in Figure 7.1.

[1] https://github.com/comunica/comunica/commit/fb1f13c0c5c6dc5c6ee7a2142d83693a58c2ed9e
[2] https://github.com/belayeng/quadstore
[3] https://www.w3.org/DesignIssues/LinkedData.html

Whilst this is not problematic in the case where *only* RDF documents need to be parsed by the engines, the architecture does not extend cleanly to the case where other types of data need to be parsed. In our case - *rules* needed to be parsed and there is a growing requirement in other applications to be able to parse other types of data such as *shape* (Knublauch and Kontokostas, 2017; Prud'hommeaux et al., 2019) files.

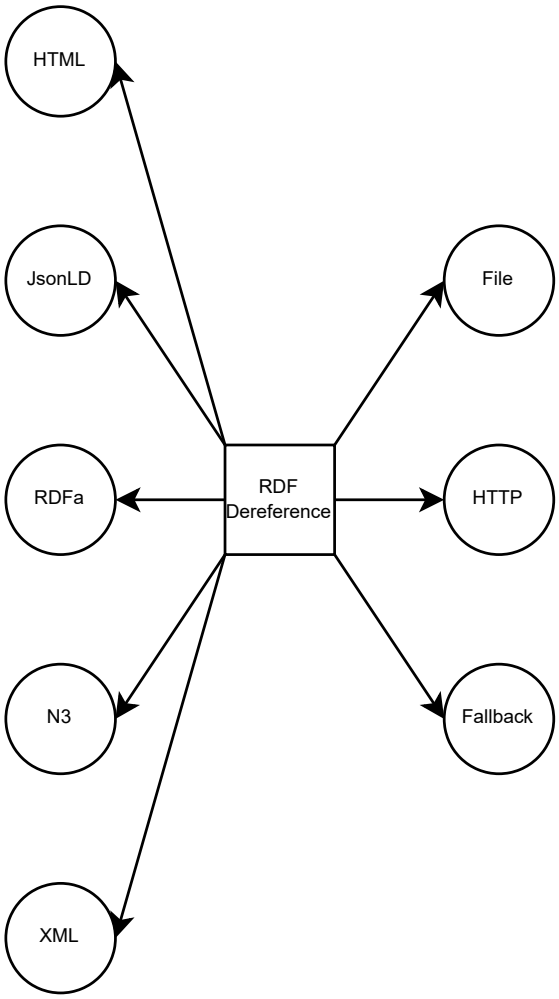

Figure 7.1: Coupled architecture for dereferencing and parsing data

Consequently, we redesigned[4] these core components of the engine to *decouple* these processes so that we would be able to cleanly add the capability to dereference and parse rules to our reasoning components. Our updated architecture is shown in Figure 7.2. This enables use to make use of the dereferencing components within T-FIRRE.

[4]https://github.com/comunica/comunica/pull/919/files

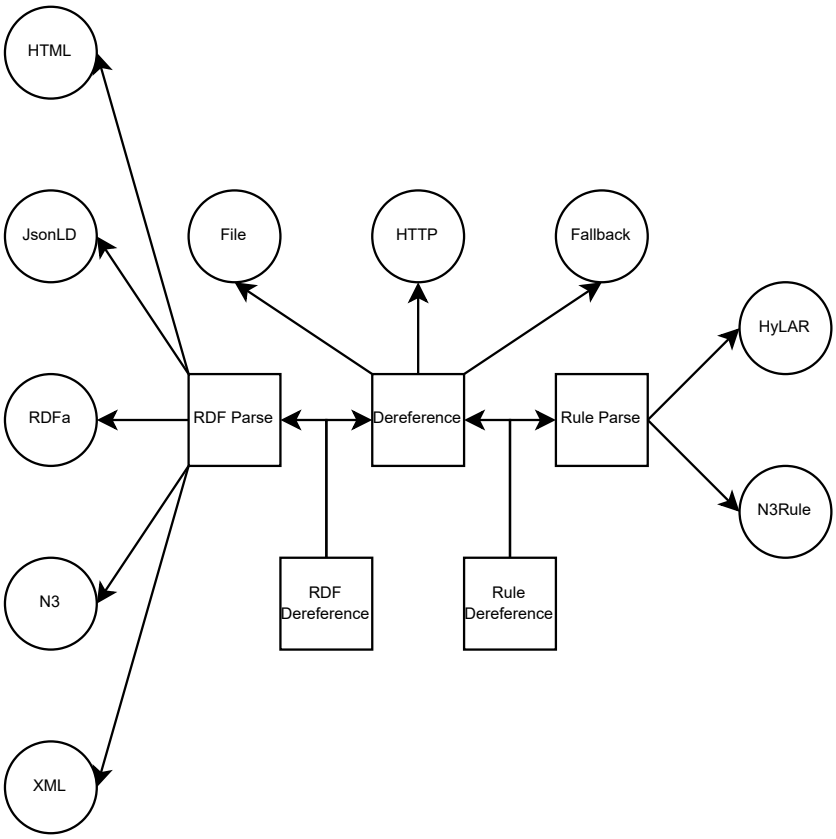

Figure 7.2: Decoupled architecture for dereferencing and parsing data

Chapter 8

T-FIRRE: Reasoning Architecture

In this chapter we describe the core T-FIRRE architecture that we have developed to enable *performant*, *interoperable* and *accessible* reasoning whilst also being *future-proof*. We first discuss the reasoning *bus* which we have defined for T-FIRRE, and how this interface can be used to enable *eager*, *lazy* and *incremental* reasoning. We then present our modular design and implementation of reasoners, rule parsers, rule optimisers and other components within T-FIRRE. Following this, we will discuss how we have designed the T-FIRRE *bus* interfaces to interoperate with the Comunica Query Engine. In particular, we will discuss the different ways of configuring T-FIRRE and Comunica to create a federated SPARQL query engine that automatically enriches SPARQL queries with implicit data. These architectures we have developed are published and in-use within Web applications.

Subsequently, we will discuss more experimental components that are still in test phases. We consider this experimental work to be well beyond the requirements for an Honours project - but have nevertheless pursued it as we believe that this is of high value to the research and development of decentralised extensions of the Semantic Web. Finally, we discuss planned future work to add further novel techniques and optimisations to the components.

R01 Interoperable Architecture and **R02 Performance** are achieved through the capacity of our T-FIRRE engine to apply reasoning in various ways and at various abstraction layers, depending on the nature of the data sources. In practice, this usually means pushing down reasoning to the lowest abstraction layer possible; for example, using index-based reasoning if the source is an N3.js store, or skipping certain parts of the reasoning process if pre-reasoning has already been applied on remote-sources.

R03 Accessibility is supported by the fact that our T-FIRRE is RDFJS compliant and interoperable with the Comunica Query Engine. This means that T-FIRRE can be used within developer tools like LDflex (Verborgh and Taelman, 2020) off-the-shelf.

R04 Future-Proof Design is supported by ensuring our design can immediately be integrated with new research and specifications around data-representation, querying, and reasoning without any changes required to the code. Examples of work that could be integrated with our reasoning components in the near-future includes:

- Link traversal research[1].
- ShapeTrees research[2].
- Data Interoperability research[3]
- The SPARQL 1.2 specification[4].
- The development of Sparqlee[5].
- Integration with the Solid Ecosystem[6].

8.1 Reasoning Bus for T-FIRRE

We wish to define a generic interface for reasoning, that is capable of handling interoperable reasoning, and is compatible with most standard techniques used for RDF reasoning. In particular we wish to be able to handle:

- **Eager Reasoning** Producing all data inferences prior to query execution.
- **Lazy Reasoning** Performing reasoning at query-execution time. Where possible, actors implementing this bus interface should be able to use information about the query to reduce the cost of reasoning. This can be done using several techniques including *backward chaining*, *query re-writing* and being *selective* about the rules/data that are used in forward chaining.
- **Incremental Reasoning** Updating the implicit facts in the database as new explicit facts are added.
- **Stream Reasoning** Reasoning over explicit data that is being continuously added to the database.

In order to achieve these outcomes we define the materialisation *bus* interface in Figure A.1. This bus is designed so that all actors implementing it, return an `execute` callback. This `execute` function returns a `Promise`[7] object when the function is called. This `Promise` *resolves* once the required operation has completed. If there is an error

[1] https://github.com/comunica/comunica-feature-link-traversal
[2] https://github.com/comunica/comunica-feature-link-traversal/tree/feature/shapetrees
[3] https://rml.io/
[4] https://comunica.dev/roadmap/
[5] https://github.com/comunica/sparqlee
[6] https://github.com/comunica/comunica-feature-solid
[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

that occurs throughout the reasoning process (including, but not limited to, a source being invalid, network errors, or a *validation rule* (Arndt et al., 2017) failure, this `Promise` will reject. We further observe that calling `execute` does **not** necessarily invoke a new reasoning operation. If the required reasoning task has already been completed then calling `execute` will immediately return a *resolved* `Promise`. If the required reasoning task is already in progress, then the `Promise` will resolve upon the completion of that in progress task. This is summarised in the state diagram in Figure 8.1.

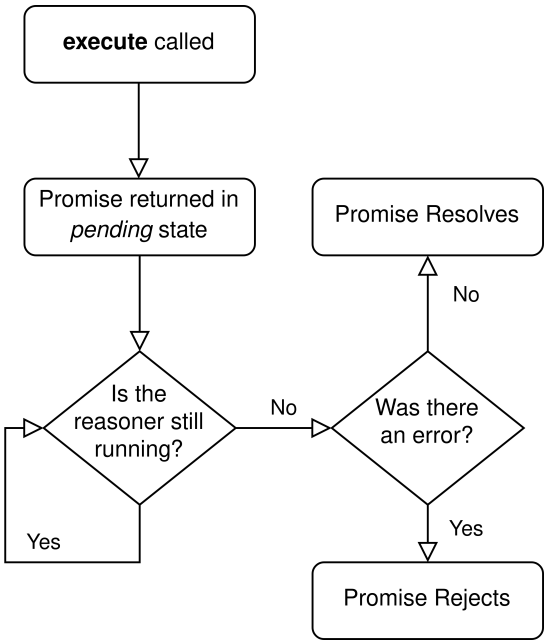

Figure 8.1: State diagram for the *bus* interface for reasoning in T-FIRRE

Reasoning Group A set of data sources that are reasoned over in union. Implicit facts are generated by reasoning over the union of all facts contained in the sources. Any two distinct reasoning groups must have a *disjoint* set of data sources, and different destinations for implicit facts. Each reasoning group *may* use different sets of rules for reasoning.

The T-FIRRE *actor*s implementing this interface are expected to satisfy the following set of semantics:

- The rules that the reasoner applies MUST be those defined in the *context*. If the actor cannot execute the rules in the context then the actor must ERROR.
- If reasoning groups are provided, it then performs reasoning between the sources *within* each group, but not between groups. If reasoning groups are not provided, it then performs reasoning between *all* sources.

  A **Reasoning Group** is set of data sources that are reasoned over in union.

Implicit facts are generated by reasoning over the union of all facts contained in the sources. Any two distinct reasoning groups must have a *disjoint* set of data sources, and different destinations for implicit facts. Each reasoning group *may* use different sets of rules for reasoning.

- All implicit facts that are materialised MUST be inserted in the Implicit Data Destination defined in the context.

  An **Implicit Data Destination** is the location where materialised data is to be stored when performing forward-chaining with T-FIRRE. As a consequence of the integration of T-FIRRE with Comunica, the implicit data destination can be any destination that Comunica generally supports for data updates. This includes files, in-memory stores and remote data stores such as SPARQL Endpoints and SOLID Pods[8].

- If a pattern is provided as input (cf. Figure A.1), then the reasoner MAY only generate those implicit facts that match that pattern.

- The reasoner only needs to materialise data which *has not already* been materialised according to the `IReasonStatus` (cf. Figure A.2). This status can define whether *all* of the implicit facts have been materialised - or whether facts matching a particular set of *patterns* have been materialised.

  The `IReasonStatus` holds metadata about the current status of reasoning within T-FIRRE. It holds information such as whether materialisation has been applied to the entire database, and whether lazy materialisation has been applied for particular *patterns*.

- If there are *updates* in the explicit data (cf. Figure A.1) provided either in the form of *insertions*, *deletions* or both - then reasoner must *add* / *delete* implicit facts to ensure that the set of implicit facts, reflects the current *explicit data* and *reason status* under the current rule-set. If a pattern is also provided as input, the actor MAY invalidate parts of the `IReasonStatus` that are not required to produce implicit data matching that pattern.

8.2 Integrating T-FIRRE and Comunica Components

The question now turns to how T-FIRRE can be integrated with the existing components in the Comunica Engine, to produce a query engine that is able to include implicit data within its results set.

[8] https://solidproject.org/users/get-a-pod

8.2.1 The *resolve-quad-pattern-reasoned* actor: Adding reasoning results to Comunica queries

Whilst the reasoning *bus* provides an interface for Comunica to trigger T-FIRRE reasoning *actions* that generate implicit data - it *does not* provide the functionality required to insert this data *back* into the results stream of a Comunica query engine.

To understand how to insert implicit data back into the results stream, we first need to recall the query evaluation process within Comunica. As discussed in Section 6.2.1, all data that is used to *evaluate* the query in Comunica is retrieved through the *resolve-quad-pattern* bus, which accepts a quad pattern as *input* and returns a pull-based iterator of the data. To enable the addition of implicit data to this iterator of results, we provide a custom T-FIRRE *actor* for the *resolve-quad-pattern* bus. This actor modifies the *context* of the query to add the *implicit* data store to the set of data sources that Comunica is to retrieve data from. After making this modification to the context, T-FIRRE will then delegate the actual task of resolving quad patterns back to the Comunica Engine.

We observe that our construction of this T-FIRRE actor enables parts of the Comunica query to be evaluated *before* all of the implicit data is materialised. This is because rather than blocking all sources, the *reasoned* quad pattern resolver in T-FIRRE, wraps the implicit destination in a `Promise`[9] which is *fulfilled*[10] once all of the relevant implicit data has been added to the destination. This means that the *federated* actor in Comunica can then immediately start pulling data from sources that have already resolved and pass that data to the query execution components. Concurrently, T-FIRRE will continue to generate implicit data while a `Promise` wraps the source that the implicit data is being added to. This `Promise` will *resolve* when T-FIRRE completes materialisation, after which Comunica will be able to read data from the implicit data source.

8.2.2 Engine configurations

Section 8.1 discussed how to materialise implicit data within a given *destination*, and in Section 8.2.1, we saw how data from the destination can be added back in the query evaluation process. The question still remains as to how the four types of reasoning mentioned in Section 8.1 are to be invoked through this interface. Figure 8.2 demonstrates the *actors* and *bus*' required to achieve invoke different types of reasoning in T-FIRRE.

1. Eager reasoning - Eager reasoning can be achieved by having the Comunica engine call the *rdf-reason* bus in `IReasonStatus` when it is instantiated by the *init-query* actor. If this is done, it will call the *rdf-reason* bus, and thus *all* inferences will be eagerly evaluated in the background before the user tries to access actor.

   When a Comunica query is eventually made, the *reasoned-resolve-quad-pattern* actor will not initiate any new reasoning processes as a consequence of the semantics of the reasoning *bus* described in Section 8.1. Rather, it will forward the implicit

[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[10] https://github.com/domenic/promises-unwrapping/blob/master/docs/states-and-fates.md

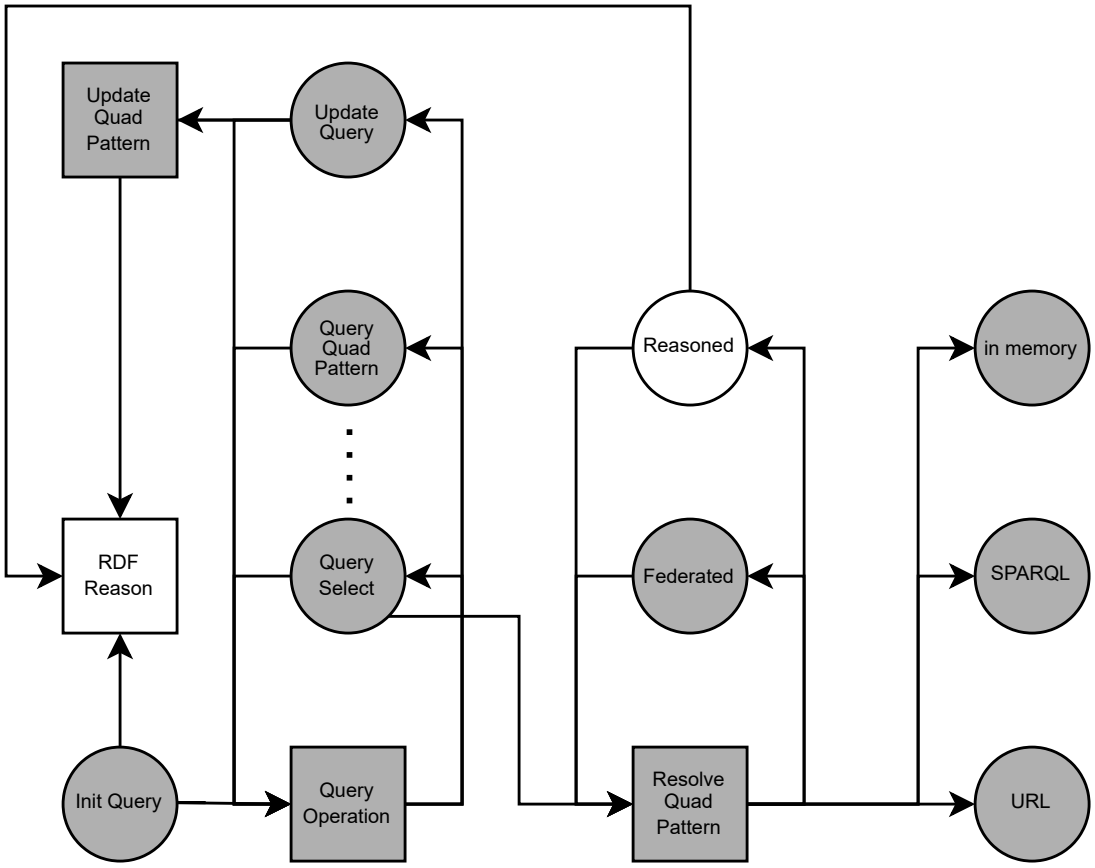

Figure 8.2: A component diagram giving the Comunica configuration required to create a SPARQL engine inference capabilities. Arrows from bus' (square) and to actors (circle) indicate actors implementing a bus. Arrows from actors to bus' indicates that the actor *can* call the bus via a mediator. The T-FIRRE components in this diagram are the *reasoned-resolve-quad-pattern* actor which is used to add implicit results back into queries, and the *RDF-Reason bus* which Comunica components can call to trigger *eager*, *lazy* or *incremental* reasoning. Arrows indicate the dependencies between components for the delegation of tasks. A query is invoked when a user of the engine triggers the `Init Query` actor.

dataset to the *federated* actor which resolves once the reasoner has finished eagerly evaluating the data.

2. Lazy Evaluation - Lazy reasoning can be achieved in by calling the *rdf-reason* bus directly from the *reasoned-resolve-quad-pattern* in T-FIRRE. The actor implementing the *RDF Reason* bus *may* choose to either evaluate *all* of the possible implicit data, or it *may* only realise the data required to produce all of the implicit facts matching a particular quad pattern.

3. Incremental/stream reasoning - Incremental views / data-streams can be maintained by being called from the *update-quad-pattern* bus, this will provide streams of any data that needs to be *inserted* or *deleted*.

4. Lazy *and* incremental/stream reasoning - In cases where the engine is only expected to produce implicit facts that match a particular set of *quad patterns* throughout its lifetime, T-FIRRE can simply choose to update views on quad patterns that have already been reasoned for in the *status*. Hence this allows incremental maintenance of lazily/partially evaluated views.

8.3 Execution Semantics

By implementing an abstract *bus* interface for reasoning, as outlined in Section 8.1 we create the opportunity for the implementation of various *reasoning* actors that *fulfil* the different reasoning requirements of the T-FIRRE *bus*.

These T-FIRRE actors could either be a stand-alone module that make use of an existing reasoner such as HyLAR[11], N3.js[12] or EYE[13] (when running in NodeJS). Alternatively, the actor can delegate various sub-tasks to other *actors* using the T-FIRRE framework.

In this section, we outline one of the most *basic* reasoning configuration that is possible using the reasoning components that we have built. The particular T-FIRRE configuration that we present in this case uses a rule-restriction algorithm to perform forward chaining on the RL reasoning profile. The configuration which we describe is summarised in Figure 8.3. A possible activity diagram for this configuration is given in Figure 8.4.

8.3.1 Resolving Rules

As we show in Figure 8.4, the first task that needs to be achieved, is collecting the rules that need to be used by the engine, which it delegates to the *rule-resolve* bus. In the default configuration that we developed, these rules can be encoded in various ways, including HyLAR[14] and N3 Rules[15] and be stored in various locations - including at

[11] https://github.com/ucbl/HyLAR-Reasoner
[12] https://github.com/rdfjs/N3.js/pull/296
[13] https://github.com/josd/eye
[14] https://github.com/ucbl/HyLAR-Reasoner
[15] https://www.w3.org/2004/12/rules-ws/paper/94/

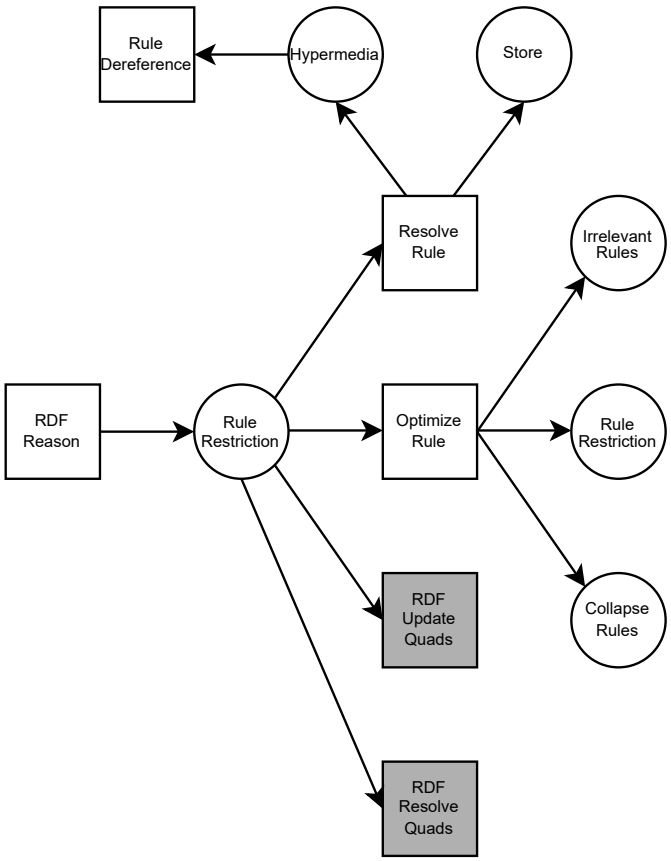

Figure 8.3: A simplified reasoner configuration that uses T-FIRRE components we have developed. This reasoner is invoked by a call to the RDF Reason *bus*. Arrows from bus' (square) and to actors (circle) indicate actors implementing a bus. Arrows from actors to bus' indicates that the actor *can* call the bus via a mediator. Bus' in grey are part of the core Comunica engine. The remaining components are part of T-FIRRE.

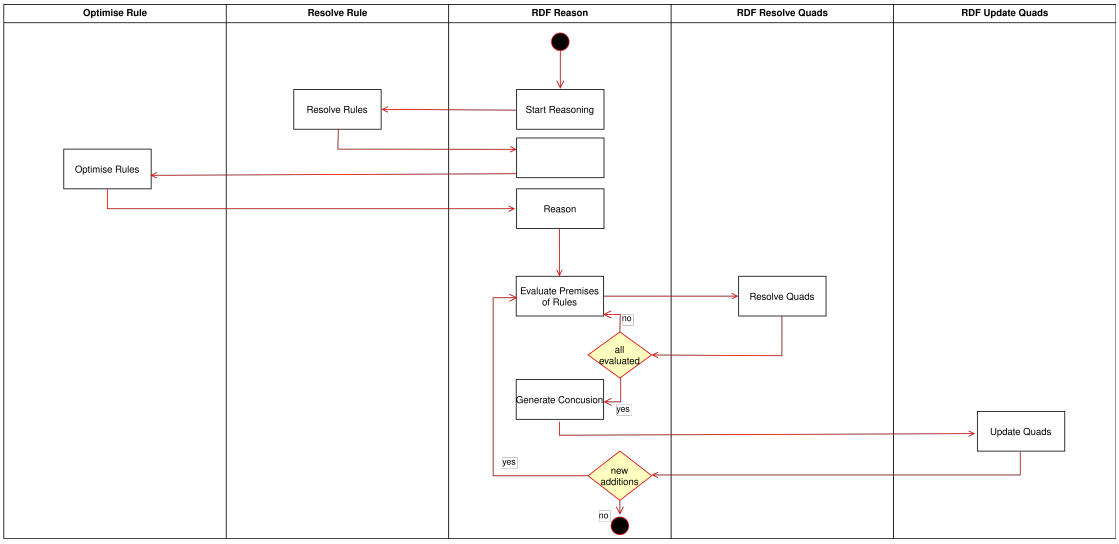

Figure 8.4: Activity flow diagram for a basic reasoner configuration with T-FIRRE

remote URLS, local files, or in an in-memory store. Implementers of the *rule-resolve* bus will generally then use a *combination* of *resolving* and *parsing* actors in order to get the rules into a format that can be used by the engine. Whilst it is not required, implementors of the *rule-resolve* bus will also usually *cache* the rules in order to speed-up subsequent requests for the rule-set - we have implemented this caching mechanism in all T-FIRRE *rule-resolve* actors we have created to date.

8.3.2 Rule Optimisation

The next key task that the T-FIRRE reasoning actor then delegates is that of rule optimisation. These optimisation actors implement many key optimisations that allow the reasoner to achieve various behaviours. These optimisation actors are generally pipelined together within the optimisation bus. Such actors that can be pipelined together to remove irrelevant rules, or optimise relevant rules. Examples of such actors include:

- Validation rules if we only want to do a data generation task (as opposed to a data validation task).
- Rules that are not relevant to the query we are doing (if we have the context of a query operation that is being performed). This is how lazy reasoning, on a forward-chaining reasoning algorithm is achieved in the *default* set-up that we provide.
- Overlapping rules (rules which will ultimately produce the same results).

Further rule optimisation actors that we have developed can:

- Collapse rules that have the same premise and/or conclusion into a single rule to prevent unnecessary re-evaluation of the same premise.
- Chain rules together to skip the production of results that are not needed in the pattern that we are matching against.
- Make rules less generic depending on the pattern that is being matched against. For instance

  `(?s, a, ?o)` ∧ `(?o, subsetOf, ?o2)` -> `(?s, a ?o2)`

  can become

  `(timbl:me, a, ?o)` ∧ `(?o, subsetOf, ?o2)` -> `(timbl:me, a ?o2)`

  if we only care about patterns of the form `(timbl:me, a, _, DEFAULT_GRAPH)`.

Chapter 9 goes into further details about the theory behind the optimisation actors that we have implemented.

8.3.3 Rule Evaluation

The T-FIRRE Rule Restriction Reasoning actor then recursively evaluates the rules using a *fixed-point rule restriction* algorithm. For this particular actor implementation, data required to check premises is retrieved using the *rdf-resolve* buses from Comunica and any new data that is found is added into the implicit data destination through the *update-quads* bus from Comunica. This destination is *also* completely customisable, and can be any update destination supported by Comunica, including SPARQL endpoints, local files, and *in-memory* stores.

This fairly naive implementation already achieves several of the research outcomes that we had set out to achieve. This includes:

- Having a generic interface to allow arbitrary reasoners to be plugged into the Comunica Engine.
- Implementing a lightweight reasoner in JavaScript, using the Comunica framework.
- Enabling lazy evaluation (on non-degenerate rule-sets) through rule optimisations that we have implemented.

8.4 Complex Execution Semantics

The *fixed-point* actor discussed in the previous section is useful for the purpose of developing an understanding of the building blocks in T-FIRRE. However, this is only a subset of the architecture that we have created - having developed various reasoning algorithms within Comunica. Many of these reasoning algorithms have more complex architectures and execution semantics in comparison to that which was outlined in Section 8.3. The advantage of this is that these actors are able to have greater modularity,

performance and expressivity in comparison to the reasoning configuration presented in Section 8.3.

Figure 8.5 presents a slightly more extensive architecture that enables us to discuss *some* of the additional functionality that is developed. The reasoning actor presented in this diagram implements a variation of the *semi-naive* forward chaining algorithm. However, unlike the standard semi-naive algorithm, this actor does not handle the task of single rule evaluations but rather delegates that task to a `ruleResolve` actor via a mediator. The `forwardChaining` function in Listing 8.1 is a pseudocode implementation of this *forward-chaining* actor. In the utility function `evalInsertRule` we can see this task delegation taking place via the `mediatorRuleEvaluate` call. `filterExisting` compenents are omitted from Figure 8.5 for the sake of diagrammatic clarity. The set of *filter-existing* actors that we have implemented in T-FIRRE are responsible for iterating over a set of quads and checking if that data already exists in the database(s). Only those quads that were not already in the database(s) are returned by implementors of this actor. We have implemented several versions of this actor so as to handle a variety of source types. We have created further related optimisation to this through the development of custom destination[16] actors to more efficiently add new facts to an implicit destination and get a stream of quads that were not previously in the store.

[16] https://github.com/comunica/comunica-feature-reasoning/tree/feat/faster-forward-chaining/packages/actor-rdf-update-quads-info-rdfjs-store

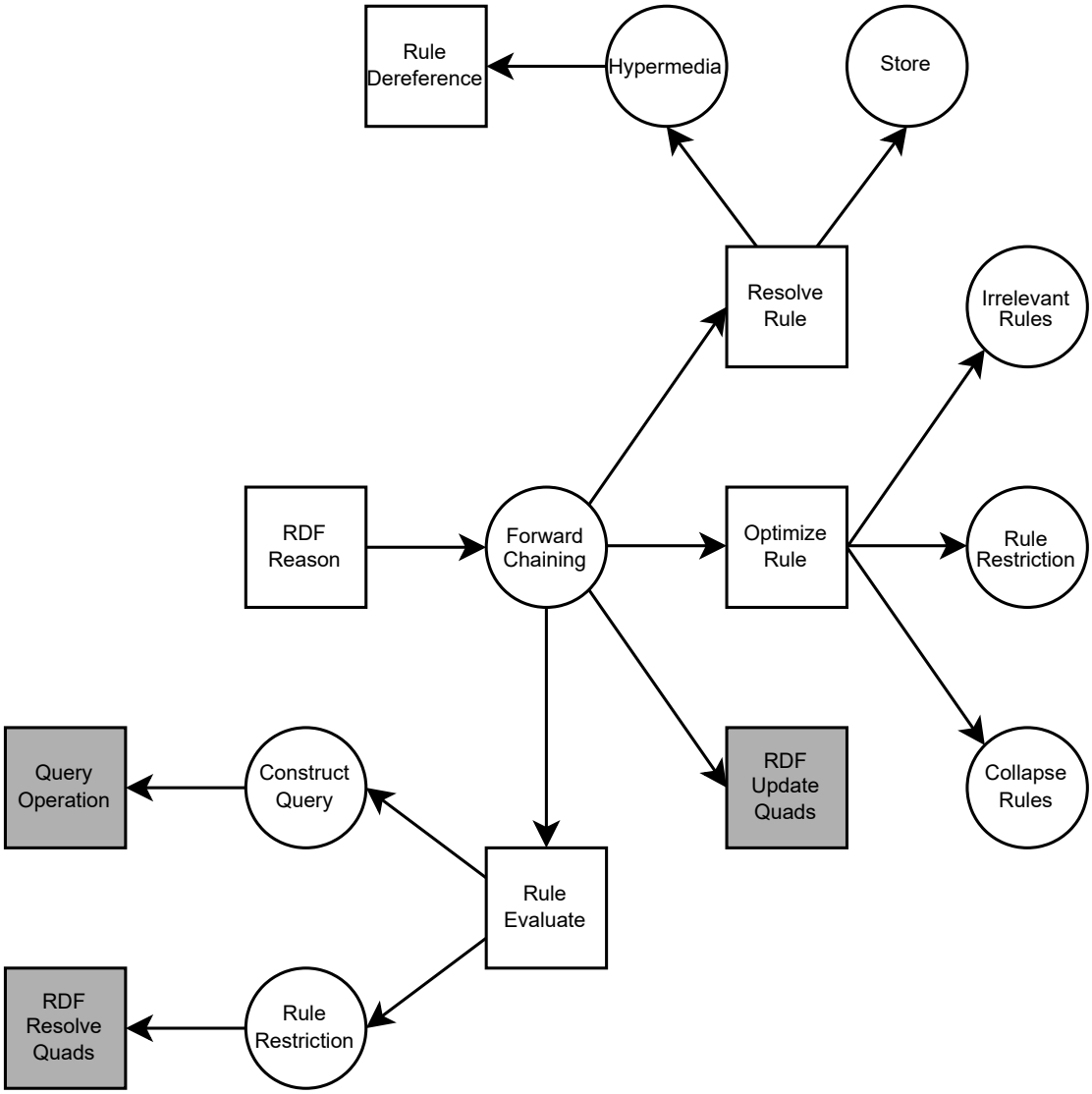

Figure 8.5: A configuration of the *forward chaining* reasoning *actor*. All components in this diagram are part of the reasoning architecture that we have developed with the exception of the RDF Update, Query Operation and RDF Resolve Quads *bus*'. These three *bus*' enable the T-FIRRE architecture that we have developed to interface with the Comunica Query Engine. This T-FIRRE reasoner is invoked by a call to the RDF Reason *bus*. Arrows from bus' (square) and to actors (circle) indicate actors implementing a bus. Arrows from actors to bus' indicates that the actor *can* call the bus via a mediator. Bus' in grey are part of the core Comunica engine. The remaining components are part of T-FIRRE.

```
# Evaluates a rule and adds new
# implicit facts to the database
# A stream of any data that was not
# originally in the database is returned
evalInsertRule(rule):

    # Delegate the task of evaluating a rule
    # to a specific rule evaluation actor
    results ← mediatorRuleEvaluate(rule)

    # Remove quads from the results that were already
    # in the dataset
    results ← mediatorFilterExisting(results)

    # Add new results to the dataset
    mediatorUpdateQuads(results)
    # Return results (implicit facts) that were not
    # already in the dataset
    return results

evaluateSubstitute(quads, rule):
    # Restrict the rule 'r' by substituting the
    # concrete values of q into the premise it
    # is expected to matched in the .next rule
    # Then evaluate the restricted
    # rule over dataset
    return evalInsertRule(substitute(r, q))
                for (r,q) ∈ quads × rule.next
                where substitute(r, q) != false

# Evaluate a set of rules over a databse using
# semi-naive forwards chaining.
forwardChaining(rules):
    # Evaluate each of the rules over the dataset
    results ← rules.map(rule: evalInsertRule(rule))

    # While new facts are still being generated
    WHILE results is NOT empty:
        R ← ∅
        FOR r IN results:
            R ← R ∪ evaluateSubstitute(quads, rules)
        results ← R
```

Listing 8.1: A forward chaining reasoning algorithm for T-FIRRE that implements the semi-naive algorithm in the `evaluateRules` function

Lets now turn our attention to the different *rule-evaluation* components that are present within Figure 8.5.

8.4.1 Delegating Rule Evaluation to Comunica

One option for *rule-evaluation* in T-FIRRE is to delegate the task of reasoning to the Comunica Query Engine. This is achieved though the implementation of a rule evaluation actor that converts a rule into a SPARQL Consruct Algebra. This algebra then gets sent from T-FIRRE to the Comunica Query Engine which will evaluate the query and return the contructed quads to T-FIRRE. This is shown in Listing 8.2.

```
mediatorRuleEvaluate(premise, conclusion):
    # Patterns for the bound-group-pattern to generate
    # the bindings for the construct query
    BGP ← ∅

    # The patterns for the bound-group-pattern
    # for the conclusion of a construct query
    C ← ∅

    FOR p IN premise:
        BGP ← BGP ∪ {quadToPattern(p)}

    FOR q IN conclusion:
        C ← C ∪ {quadToPattern(q)}

    # Convert the set of BGPs to a SPARQL BGP algebra
    BGP ← createBGP(BGP)

    # Create the SPARQL algebra for a construct query
    # that corresponds to the evaluation of the input rule
    operation ← createConstructQuery(BGP, C)

    # Evaluate the generated SPARQL algebra using the
    # query operation actors in the Comunica Engine
    return runQueryOperation(operation)
```

Listing 8.2: A rule evaluation actor that delegates the task to the Comunica Query Engine via a Construct Query

8.4.2 Direct rule evaluation with Dynamic Nested Inner Joins

Another option, rather than delegating the task of rule evaluation; is for the actor to implement it. For RDFS style rules this is relatively straightforward to implement.

This is because it can be achieved by performing a dynamic nested inner join over the premises of a rule to generate the set of all possible mappings from variables ocurring in the rule, to concrete values. We show how this can be done in Listing 8.3. As shown in Listing 8.3 this algorithm only needs to call upon one external set of actors, which are the *rdf-resolve-quad-pattern* actors used to resolve data from different sources.

```
mediatorRuleEvaluate(premise, conclusion)
    # M contains a single empty dictionary
    M ← {{}}

    # Generate all possible permutations of mappings
    # from variables to concrete values for the given
    # premises in a rule. This is calculated using a
    # dynamic inner join
    FOR premise IN rule.premise:
        N ← ∅

        FOR m IN M:
            # Substitute any variables in the premise with a
            # concrete value defined in the mapping m if
            # a concrete value is defined.
            pattern ← substitute(premise, m)

            # Retrieve all quads that match a pattern
            # using the actors within Comunica
            quads ← mediatorRdfResolveQuadPattern(pattern)

            # Create a map, L which maps all of the variable
            # terms in the pattern, to the concrete value
            # in the quad
            FOR q IN quads:
                # L is an empty dictionary
                L ← {}

                FOR key, term in q:
                    IF pattern[key].termType = 'Variable':
                        L[pattern[key].value] ← term

                # Merge the L and m dictionaries and add
                # the result as an element of N
                N ← N ∪ ({ ...L, ...m })

        M ← N

    # Generate new inferences by substituting the mappings
    # into the conclusion of the rule
    return subtitute(c, m) for (c, m) ∈ conclusion × M
```

Listing 8.3: A rule evaluation actor that explicitly evaluates RDFS rules by performing a dynamic inner join on premises; and then generates conclusions. For algorithmic simplicity we ignore the case where the same variable occurs twice within the same premise

8.5 Summary

In this section we have presented the Interoperable T-FIRRE Reasoning engine. The configurability of this tooling, in addition to its ability to integrate with the existing Comunica Engine, indicates that that we have indeed developed an **R01 Interoperable Architecture**. This **R01 Interoperable Architecture** enables us to create many use-case customisations that can result in a good **R02 Performance** for reasoning; whilst sill having abstract reasoning algorithms to fall back on. As noted in the introduction, **R03 Accessibility** is supported by the fact that our T-FIRRE is RDFJS compliant and interoperable with the Comunica Query Engine. This means that T-FIRRE can be used within developer tools like LDflex (Verborgh and Taelman, 2020) off-the-shelf. **R04 Future-Proof Design** is, again, supported by the fact that our design can immediately be integrated with new research and specifications around data-representation, querying, and by simply re configuring the engine components.

Chapter 9

Theoretical developments for RDF Reasoning on a Decentralised Web

This chapter explores some common challenges and opportunities arising in reasoning over a decentralised Web environment. Many of these algorithms are tailored towards a specific set of use case(s). The T-FIRRE architecture on which we implement these algorithms is conducive to such constraints. This is because the different *actors* can implement algorithms that are tailored for each use case and then the engine can *mediate* for the best actor to use based on the conditions that the engine is facing at run-time.

Consequently, the work in this section explains some of the improvements in **R02 Performance** of our reasoning architecture. Often, this is done by exploiting the modular architecture that we have created to apply custom *optimisations* and *reasoning algorithms* based on the current reasoning task.

In Section 9.1 and Section 9.2 we discuss reasoning algorithms that we have designed and implemented within the reasoning engine that we have developed.

Sections 9.3, 9.4 and 9.5 then outline the various future optimisations that can be implemented with the engine. For the most part, these optimisations are partially implemented and awaiting some blocking changes in the core of Comunica in order to be completed in T-FIRRE.

9.1 Lazy Reasoning

This section explores a novel technique we have developed to improve reasoning performance. The goal is to enable lazy (or partial) forward-chaining for RL profile reasoning. The *intuition* and *implementation* for this is rather simple - if we know a particular set of *quad patterns* that we need to match against in a query, then we can restrict the *rules*

that are applied to our data, such that only those rules that are *required* to produce all implicit facts matching said patterns are kept.

This algorithm operates by starting with a list of quad `pattern`(s) that we need to match against, for a query. Then, it searches through the set of all rules; testing each rule to see if any conclusion from that rule could possibly produce a quad that matches one of the `pattern`(s). Any rule that tests `true` gets added to the set of `restricted` rules that will be passed to the reasoner. The premises of any rules added to the `restricted` set, get added to the set of quad `patterns` that need to be matched against. This process repeats until no new rules get added to the `restricted` set. A naive algorithm for this is given in Listing 9.1 where the restricted set is denoted by `R`.

```
restrictNaive(rules, patterns):
    R ← ∅

    WHILE R has not reached a fixed point:
        FOR premises, conclusions IN rules:
            FOR pattern IN patterns:
                IF ∃(c, p) ∈ conclusions × patterns s.t. c matches p:
                    R ← R ∪ {r}
                    patterns ← patterns ∪ r.premise

    RETURN R
```

Listing 9.1: A naive algorithm to restrict forward chaining rules to only produce implicit facts for required patterns.

Unfortunately, we note that the RDF/RDFS entailment rules (given in Listing A.1) are a *degenerate* case for lazy, *forward-chaining* reasoning of this style; as the presence of unrestrictive patterns (that is patterns of the form `(?s, ?p, ?o)`) means that without any knowledge of the data, we cannot remove any rules prior to performing materialisation. An example of this unrestrictive pattern can be seen in the first RDFS rule in Listing A.1.

However, this technique still holds promise for applications that use custom rule-sets; for instance, particular examples of schema alignment.

This lazy reasoning algorithm is included as a rule optimisation actor in the T-FIRRE architecture that we have released[1].

[1] https://github.com/comunica/comunica-feature-reasoning/tree/master/packages/actor-optimize-rule-pattern-restriction/lib

9.2 Further Rule Optimisations

Next, we outline several more novel rule optimisations that we have implemented as *independent actors* within T-FIRRE. By implementing them as separate actors, developers using T-FIRRE are able to enable/disable these optimisations at will. The enabled optimisations will have the rules pipe-lined through them.

Importantly, we observe that the design of these optimisations makes the rule-set output from the optimisations *independent* of the order in which the rules were pipe-lined through the optimisation actor.

Each of the actors discussed is either released as a package - or implemented in an experimental branch[2] of the reasoning repository.

9.2.1 Rule Substitution

One possible way to extend the Lazy Reasoning optimisation that is proposed in Section 9.1, is through the use of *rule-substitutions* based on the quad-pattern that is being matched against. This optimisation follows a similar principle to backward chaining - but instead is used to create *more precise* rules that further reduce the patterns that need to be retrieved during the reasoning process. The consequence of this is that the network load required for the reasoning process is reduced if reasoning against remote sources; and similarly, the amount of explicit data that needs to be searched over when reasoning over in-memory data is also reduced. This algorithm is given in Listing 9.2.

```
restrictSubstitution(rules, patterns):
    R ← ∅

    WHILE R has not reached a fixed point:
        FOR premise, conclusion IN rules:
            FOR pattern IN patterns:
                r ← substitute(premise, conclusion, pattern)
                IF r is defined:
                    R ← R ∪ {r}
                    patterns ← patterns ∪ r.premise

    RETURN R
```

Listing 9.2: An rule substitution algorithm to restrict forward chaining rules to only produce implicit facts for required patterns

We have *implemented*[3] an experimental version of this within T-FIRRE.

[2] https://github.com/comunica/comunica-feature-reasoning/tree/experimental-development
[3] https://github.com/comunica/comunica-feature-reasoning/tree/next/major/packages/actor-optimize-rule-pattern-substitution

9.2.2 Stripping Validation Rules

There are many cases in which validation rules within RDF reasoning are not required (Gayo et al., 2017), especially when independent validation procedures involving SHACL (Knublauch and Kontokostas, 2017) or ShEx (Prud'hommeaux et al., 2019) form a part of the data pipeline. This is because the RDF validation rule will end up 'duplicating' the validation operation performed by the SHACL (Knublauch and Kontokostas, 2017) or ShEx (Prud'hommeaux et al., 2019) constraints - making the validation rule redundant. An example of such a validation rule is given in Listing 9.3. The equivalent SHACL expression of this validation rule is given in Listing 9.4.

```
(?x a http://www.w3.org/2002/07/owl#Nothing) -> false
```

Listing 9.3: Example an RL profile validation rule

```
ex:NothingShape a sh:NodeShape ;
    sh:targetClass owl:Thing ;
    sh:property [
        sh:path a ;
        sh:not [
            sh:hasValue owl:Nothing ;
        ] .
    ] .
```

Listing 9.4: Validation rule for `owl:Nothing`

We have implemented this validation rule stripping[4] in the current alpha release of T-FIRRE. The implementation of this is a simple filter given in Listing 9.5

```
optimize(rules):
    R ← ∅

    FOR r IN rules IF r.conclusion != false:
        R ← R ∪ {r}

    return R
```

Listing 9.5: Algorithm for removing validation rules

9.2.3 Reconciling and Nesting Premises

Another straightforward optimisation that we implemented within T-FIRRE, was to identify and collapse rules that had *semantically equivalent* premises. A performance

[4] https://github.com/comunica/comunica-feature-reasoning/tree/master/packages/actor-optimize-rule-remove-false-conclusion

improvement can be obtained on such rules by collapsing them into a single rule, thus preventing duplicate computation of the shared premise. The algorithm given in Listing 9.6 prescribes how to collapse these premises.

```
optimize(rules):
    # Normalise rules so so that the ordering of premises
    # and choice of variable names is consistent
    R ← normalize(rules)
    N ← ∅

    FOR r IN R:
        FOR i in N IF i.premise = r.premise:

            IF i.conclusion = false
                OR i.conclusion = false:
                    # If either is a validation rule
                    # then we only need to produce false
                    # as this causes a reasoner to error
                    # on invalid data
                    i.conclusion ← false
            ELSE
                # Otherwise the conclusion is the union
                # of all the conclusions of the rules
                # that share the same premise
                i.conclusion ← i.conclusion ∪ r.conclusion

            BREAK
        ELSE
            N ← N ∪ {r}
    RETURN N
```

Listing 9.6: Algorithm for reconciling premises

We now illustrate the outcome of this algorithm for the rules given in Listing 9.7. We observe that both rules share the premise `?u a rdfs:Class`. Hence, the rule can be reduced to the single rule given in Listing 9.8. While the resultant rule given in Listing 9.8 is not a Horn Clause as a result of having a conjunction in the conclusion; it can still be handled by the reasoning components that we have developed for Comunica and hence is a valuable optimisation to make.

```
(?u a rdfs:Class) -> (?u rdfs:subClassOf ?u)

(?u a rdfs:Class) -> (?u rdfs:subClassOf rdfs:Resource)
```

Listing 9.7: Example of RDFS rules with the same premise

```
(?u a rdfs:Class) ->
    (?u rdfs:subClassOf ?u) ^
    (?u rdfs:subClassOf rdfs:Resource)
```

Listing 9.8: Example of RDFS rules with the same premise

A more complex case can be seen in Listing 9.9 where only a subset of the premises in the rules overlap. We introduce a nested rule algorithm in Listing 9.10 to handle these types of rules.

```
(?u rdfs:subClassOf ?x) ^ (?v rdf:type ?u) -> (?v a ?x)

(?u rdfs:subClassOf ?v) ^ (?v rdfs:subClassOf ?x)
                                -> (?u rdfs:subClassOf ?x)
```

Listing 9.9: Example of RDFS rules with a partially shared premise

In Listing 9.10 we give the algorithm to generate a set of nested rules given a set of unnested rules. This algorithm first normalises the rules so that the order of the premises in a rule is deterministic. The algorithm then iterates through each of the rules to create a set of *tree* representations of the rules. Where each set of rules that share a premise share a root, rules that share a second premise share a branch and so forth. The rules 'branch out' when they come across a premise at depth n that does not match.

If there is a validation rule, then no branches can extend beyond where the final premise of the validation rule occurs. Thus, if a set of premises exist that match all of the premises of the validation rule, then the validation rule will cause a false result to be produced and the reasoning procedure will *terminate*. Consequently, there is no point in extending rules beyond this point.

The benefit of structuring rules in this way is similar to that of the shared premise optimisation that we introduced in Listing 9.6 - in that it enables *internal* optimisations within the reasoning engine by *skipping* the repetition of `match` and `join` operations across rules that share a similar set of premises. In turn, this enables a performance improvement within T-FIRRE.

```
optimize(rules):
    # Normalise rules so so that the ordering of premises
    # and choice of variable names is consistent
    R ← normalize(rules)
    N ← ∅

    FOR r IN R:
        L ← N

        FOR p IN r.premise:
            FOR i in L IF i.premise = p:
                l ← i
                BREAK
            ELSE
                l ← { premise: p, next: ∅, conclusion: ∅ }
                L ← L ∪ {l}

            # If we have reached the point of a
            # validation rule then we can terminate
            IF l.conclusion = false:
                BREAK


            IF p is last element of r.premise:
                # If the rule we are adding is a validation
                # rule then remove anything that comes after
                # it
                IF p.conclusion = false:
                    l.conclusion ← false
                    l.next ← ∅
                ELSE
                    l.conclusion ←
                        p.conclusion ∪ l.conclusion
            ELSE:
                L = l.next

    RETURN N
```

Listing 9.10: Algorithm for generating nested rules

The algorithms described in this section are implemented, and currently in testing stages[5]

[5] https://github.com/comunica/comunica-feature-reasoning/tree/next/major/packages/actor-optimize-rule-reconcile-premise

within the T-FIRRE engine.

9.3 Federated Reasoning Against Large Databases

A requirement that we have identified in the world of linked data[6] is the ability to perform federated reasoning against a large database (for instance DBPedia) and a smaller document - such as a personal profile document. In many cases, the larger database has *already had* inferences, such as RDFS rule inferences, materialised[7] (Sima et al., 2019). This is a fact that can be readily exploited, so as to reduce the computation that is required by the client.

9.3.1 Overview of Optimisation

The operation of federated reasoning over a knowledge base and a personal profile document is equivalent to applying *incremental reasoning*, with the pre-reasoned database as the original knowledge base and the profile document are the facts that are to be inserted.

Consequently, we can just apply the incremental variant semi-naive algorithm (Abiteboul et al., 1995) (which is just the semi-naive algorithm without the first step of complete materialisation).

We have not implemented this optimisation. This is because this optimisation can be implemented using the existing actors for reasoning with *semi-naive* algorithm and is consequently just a development task, rather than a research problem.

9.3.2 Detecting Reasoning Profiles in Remote Knowledge Graphs

To enable the above optimisation, we first need to know which reasoning profiles have indeed been applied to a database in advance. This work aims to provide a demonstrative use case, as to why linked data sources should publish metadata about the reasoning profiles that have been applied in advance; and thus promoted so that such metadata can be published in standards such as SPARQL 1.2.

We further suggest that as part of the SPARQL 1.2 standard, one should have the option to add metadata parameters to the request that a *user makes*, which can specify the preferred rule profiles they wish to have applied to their results (if any).

There would then be a *bus-rule-detect-profile* and an *actor-rule-detect-profile-metadata* sitting on top of this, which detects the rule profile based on the metadata revealed by the source.

[6] For discussions on this topic see; https://github.com/comunica/comunicafeaturereasoning/issues/23, https://github.com/SolidLabResearch/Challenges/issues/14

[7] In the case of the Virtuoso backed DBpedia database, these inferences are not provided by default, but can be requested via custom SPARQL Pragmas http://vos.openlinksw.com/owiki/wiki/VOS/VirtSPARQLReasoningTutorial

In the interim, we need to find other means to detect reasoning profiles that have been applied to data sources. We do this by introducing a T-FIRRE actor *actor-rule-detect-profile-quad-pattern-heuristic*; which checks for certain quad patterns that are typically not stated as explicit data in databases. An example of this, is any data conforming to the pattern `?s a rdfs:resource`; as users do not typically specify the class `rdfs:resource`, but triples of that form are produced by the RDFS inference scheme.

9.3.3 Open Questions

In the algorithms and implementation that are described above, we have made the *strong* assumption that any 'pre-reasoned' databases have applied an *identical* set of rules to the rules which we wish to use for the federated query. This assumption does not hold true in general, with databases possibly applying a subset, superset, or otherwise disjoint set of rules; to the ones which are being applied to the federated query.

In the case of a *subset* of the rules of the federated query being applied to the remote (or in-memory) database, there are several possible approaches that can be taken so as to exploit the work that has already been done by the database. One such approach is to create a *diff* of rules, which have not yet been applied to the remote source and only applied those rules in the first step of the semi-naive (Motik et al., 2014) algorithm[8].

There remains the question of how to handle implicit data within the remote source that would not have been produced by the rule set in use. Semantically, we need to remove the data that is produced by rules which we do not want to have applied, to produce results that are sound with respect to the rules that we input into our federated engine. However, most servers do not provide the metadata that is required in order to be able to distinguish between data that is explicit and implicit within the store.

Moreover, it is impossible to determine whether any fact within a database is implicit with complete certainty, as any fact which *would* have been generated through a process of reasoning may have already been placed into the dataset as an explicit fact, prior to the application of a reasoner. For instance, if we have the single rule `?s ?o ?o -> ?s a owl:Thing` applied to a KG; and the view of the data with reasoning applied is

```
ex:Jesse foaf:knows ex:Armin .
ex:Jesse a owl:Thing .
```

then there is no way of knowing whether the explicit data (that is the data in the KG prior to reasoning) was

```
ex:Jesse foaf:knows ex:Armin .
```

or

[8] https://github.com/comunica/comunica-feature-reasoning/issues/23

```
ex:Jesse foaf:knows ex:Armin .
ex:Jesse a owl:Thing .
```

9.4 `owl:sameAs` Handling

The `owl:sameAs` predicate is used to identify when two URIs (Berners-Lee et al., 1998) *semantically* represent the same entity. For instance, the triple `(timbl:me, owl:sameAs, wdt:Q80)` is used to represent the fact that the identifiers of *Tim-Berners Lee* on his personal profile card and on Wikidata, represent the same entity.

The *semantic* consequence of this `owl:sameAs` relationship, is that every statement about `timbl:me` is also a valid statement about `wdt:Q80` and vice-versa. If there are many `owl:sameAs` relationships in the database(s) that one is performing reasoning over, this can lead to an exponential explosion (Motik et al.) in the size of the dataset when reasoning is applied, if one naively re-applies the same materialisation to all of the entities. Not only can the application of this `owl:sameAs` rule slow *materialisation* by an order of magnitude - it has also been shown that the application of `owl:sameAs` rules can increase the size of a database by a factor of up to 7.8x when loading in real, publicly available data sets (Motik et al.).

Motik et al. proposes a solution to this problem in a single database case, through a procedure known as entity re-writing. In this approach, a single representative of all *equivalent* entities is chosen to be used within the database and all statements are made with respect to that entity. The database also stores a record of all *equivalent* entities and hence, adds these equivalent entities back into the *results* when a query is made over the database. However, the solution proposed in this single database case cannot be directly applied in a distributed setting, in which we often do not have write access to remote databases. Even if we did, it is undesirable to run the entity re-writing algorithm over a network where it is likely to be highly underperformant.

Hence, we propose[9] the following solution to enable handling of the `owl:sameAs` relationship in a distributed data setting. The idea that we propose', *adapts* the concept of *re-writing*, by using a single representative entity within the *internals* of T-FIRRE - whilst recognising various representations of the entity when *interfacing* with other data-sources.

Similar to the centralised case, we propose that each *reasoning group* has its own record of *equivalent entities*. In order to handle `owl:sameAs` reasoning on these distributed sources, we must maintain a record of equivalent entities. This record is then used to canonicalise entities coming in from remote sources and 'expand' the data back to represent all entities, as it exits the reasoning internals and goes into the query-evaluation engine.

[9] This proposal is future work as it is wise to wait until there is shared dictionary support in the RDFJS ecosystem as per https://github.com/rdfjs/N3.js/issues/289

We observe that this expansion is *required* to happen prior to query evaluation (rather than just expanding the final bindings that are evaluated for the query), in order to correctly maintain SPARQL semantics (Motik et al.).

Maintaining a record of equivalent entities : The challenge in a federated reasoning context, is that of creating and maintaining a record of equivalent entities whilst minimising the network overhead of looking up the equivalence relations. We propose a solution in which T-FIRRE will fill the record of equivalent entities, by *detecting* the equivalents of a particular entity when looking at a particular source - and only do this for entities that we *care* about in the lazy reasoning settings. The execution semantics for this are as follows:

1. Whenever quads are being streamed in from a remote source, we check for entities within the stream that are not currently in the *equivalence record*. If such an entity exists, then we need to lookup its equivalence. If the source we are looking at *is known to have reasoning applied to it*, then we can simply lookup all equivalent entities within the source using the pattern `(ex:newEntities, owl:sameAs, _)`. If reasoning that uses `owl:sameAs` semantics is *not* applied to the source, then this check will need to be recursively repeated for any newfound equivalents in the database, until all equivalents are found. Following this, any newfound entities, then need to be looked up in *all other* remote sources to see if there are any other equivalence relations. This process is recursively repeated until all members of the equivalence class of entities are found.

2. As quads are streamed in from a remote source, they are converted to use a chosen *canonical* representative for each equivalence class of entities using the equivalence map.

3. Conversely, whenever a *resolve-quad-pattern* action is made against a source, T-FIRRE must issue a *resolve-quad-pattern* call with *each* combination of representatives in the pattern and take the union of the results; *unless* the source already has reasoning over `owl:sameAs` patterns - in which case only the original pattern needs to be used.

4. As results are streamed *out* of the *resolve-quad-pattern-reasoned* actor which obtains data from all sources and the implicit source with entities in *canonicalised form*; the results must be *expanded* back out to have one quad for each representative, each time the canonical representative for an *equivalence class* of entities is seen.

9.5 `owl:subClassOf` Handling

The semantics surrounding `owl:subClassOf` and `owl:subPropertyOf` can trigger a similar 'explosion' in implicit data when doing forward chaining materialisation.

However, solving this challenge is more difficult than that of `owl:sameAs`, due to the

fact `owl:subClassOf` is only a transitive relation and not an equivalence relation. This means that instead of the record structure proposed in Section 9.4, this use-case requires data-structure that is required is a doubly linked list, where each node in the list adheres to the interface

```
interface SubClassNode {
    classes: RDF.Term[];
    next: SubClassNode;
    prev: SubClassNode;
}
```

and for each is also a record, mapping each class to its position in the subclass/sub-property hierarchy

```
type HeirarchyMapping = Map<RDF.Term, SubClassNode>
```

Chapter 10

Optimisation of Data Structures for Federated Execution

Non-blocking, dynamic, pull-based iterators are the core data structure backing many query and reasoning engines. This includes the Comunica Engine and the T-FIRRE engine that we have developed. The work outlined in this section consists of optimisations that we have made to the AsyncIterator data structure. These performance improvements that we have made are non-trivial, resulting in performance improvements in the architecture that are in the orders of magnitude.

Since all data that is being *reasoned* and *queried* over within T-FIRRE and Comunica passes through this data structure, it is *critical* that this data structure is performant as possible.

Without discovering the performance deficiencies in this data structure and our resultant work to resolve the deficiencies, we would not have achieved the performance in T-FIRRE that is observed in our results section.

Section 10.1 provides background material required to understand the `AsyncIterator` data structure that we optimise throughout the remainder of the chapter. Sections 10.2 to 10.6 outline performance deficiencies that we identified in the data-structure in addition to the solutions that we, the author, proposed. In Section 10.2 and Section 10.3 we collaborated with those acknowledged at the start of this thesis to implement the proposed solution. In Section 10.4 and Section 10.5 we, the author, implemented the entirety of the proposed solution.

10.1 Background

In reasoning and query architectures such as T-FIRRE and Comunica, execution plans are formed from a *tree* of iterators which data is pulled through as required (Verborgh

et al., 2016). As such, the data structures used to create these pull-based iterators must be as efficient as possible; otherwise, significant performance degradations[1] may be observed in the engines that make use of these iterators.

The AsyncIterator[2] package was initially created to support the first query engine capable of evaluating queries over linked data fragments (Verborgh et al., 2016). The package itself added what was a missing piece in the NodeJS architecture at the time, namely, a data-structure that can 'return multiple asynchronously and lazily created values'[3]. Despite experimental support for AsyncIterators now emerging within the core NodeJS ecosystem, they have significant performance issues[4] at the time of writing; in comparison to the AsyncIterator package (after our work on improving the internals of the data-structure).

10.1.1 Implementation of AsyncIterator

The `AsyncIterator` class implementation extends the NodeJS `EventEmitter` class. The data from a `AsyncIterator` can be consumed in two ways. The first way of accessing data is *on demand*. In this mode, an element is read from the iterator by calling `.read` method. This method can either return the next element from the iterator, or `null` if no new items are available. The iterator emits a `readable` event when new items are available. All `AsyncIterator`s must implement this `.read` method.

The second way of accessing data is in *flow mode*. In *flow mode* users of the API access data by subscribing to the `data` event. Once the data event is subscribed to, items will immediately be emitted as they become available; rather than being lazily `.read`. This flow mode is generally implemented by continually calling the `.read` method until no new items are available.

In both modes, the iterator emits an `end` when no new elements are available.

10.1.2 Achieving a non-blocking behaviour

The reason that the `AsyncIterator` is able to achieve non-blocking behaviour (unlike the synchronous `Iterator` in the NodeJS API) is because the `.read` method is able to return `null` at any time, which indicates that items are *not currently* available, but *does not* indicate that the iterator has ended. By having a `readable` event to indicate when data again becomes available other blocking patterns such as repeatedly calling `.read` until it returns new elements can also be avoided.

[1] https://github.com/comunica/comunica/issues/676
[2] https://github.com/rubenverborgh/asynciterator
[3] https://github.com/rubenverborgh/asynciterator
[4] https://github.com/nodejs/node/issues/42618

10.1.3 Achieving a 'Tree' behaviour

To execute most query and reasoning plans - we need to achieve a tree-like pipeline rather than a single iterator pipeline. In practice, this is achieved by `cloneing` an iterator at any point at which the iteration tree branches. This results in a new iterator per 'branch' in the tree.

The primary place that such *branching* behaviour is seen within the Comunica query engine is within *update queries*. In update queries, bindings (results) get passed through a map that detects any insertions/deletions. Any insertions/deletions are then piped into the relevant destination.

Within T-FIRRE the primary place that *branching* occurs is when evaluating rules with shared premises.

10.2 The BufferedIterator

The `BufferedIterator` extends the `AsyncIterator` class and is designed to enable items with an expensive, or asynchronous, generation processes to be created and buffered prior to being `.read` so that items can be immediately available for use when the iterator is needed. Once the buffer is non-empty, the iterator becomes `.readable`, and elements can be read from it.

However, the design of the iterator can result in a significant slowdown[5] when the iterator is being used for fast, synchronous item generation. This is because asynchronous calls are made whenever the buffer fills in order to prevent blocking the execution of the NodeJS event loop[6].

Furthermore this iterator implements the `.read` method by calling `.shift` on the internal data buffer - this is despite the `.shift` method having a slow $O(n)$ implementation (with respect buffer length) within the V8 engine[7].

We have proposed solutions to each of these problems. For the former problem around asynchronous calls, we implemented a complete solution that resolves this, as well as several API issues within the `AsyncIterator` library[8].

The latter problem that we discovered[9] was resolved by modifying the iterator to be backed by linked-list[10].

[5] https://github.com/RubenVerborgh/AsyncIterator/issues/44
[6] https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
[7] We have also raised this issue with the Chrome team in https://bugs.chromium.org/p/v8/issues/detail?id=12730
[8] This work can be seen at https://github.com/RubenVerborgh/AsyncIterator/pull/45 - it has not yet been merged as it contains breaking changes, and we want to create a few more minor releases with non-breaking changes first.
[9] https://github.com/RubenVerborgh/AsyncIterator/issues/38
[10] https://github.com/RubenVerborgh/AsyncIterator/pull/46

Table 10.1: Applying 10 transformation to 500,000 elements with a `BufferedIterator` before and after switching to a linked-list buffer.

Before (ms)	33900
After (ms)	70
Speedup	484.2x

This work resulted in a 484x speedup on common use-cases such as applying ten buffered transformations to 500,000 elements. The results of this experiment are shown in Table 10.1.

10.3 Wrapping Sources

The 'AsyncIterator' package exports a `wrap` method which allows any object implementing the `EventEmitter`[11] interface to be accessed using the `AsyncIterator` API. The original implementation of this, involved placing the *wrapped* `EventEmitter` into *flowing mode*[12] and placing all of the emitted elements into a buffer which can then be pulled through from the iterator.

We identified that this is *sub-optimal* in many cases where the element being wrapped implements a *superset* of the `EventEmitter` interface that supports pull-based iteration. A key example of this is an N3.js store[13].

In the original implementation of `wrap`ping, this means that if the `match` function is called on the N3.js store, then *all* of the *matches* would placed into *buffer*; *then* the iterator will become *readable* and allow elements to be pulled. This results in both a *performance degradation* and unnecessary use of memory.

Consequently, we recommend[14] that this behaviour be fixed to instead wrap the *pull* method of the upstream iterator and skip buffering.

We ran experiments to test the performance of reading all data from a `wrap`ped pull-based iterator before and after these changes. These experiments were tested both on iterators that were wrapped in a `Promise` and those that were not. These experiments read 200,000 elements from a `wrap`ped `ArrayIterator`. The results of these experiments can be seen in Table 10.2.

10.4 UnionIterator

Another commonly used iterator within the library is the `UnionIterator` which takes a set of `AsyncIterator`s as input, and then enables *union* of the elements from the

[11] https://nodejs.org/api/events.html#class-eventemitter
[12] https://nodejs.org/api/stream.html#two-reading-modes
[13] https://github.com/comunica/comunica/issues/965
[14] https://github.com/RubenVerborgh/AsyncIterator/pull/63

Table 10.2: Time taken to read 200,000 elements from an `ArrayIterator` with four applications of the `wrap` (No Promise), and with four alternating applications of `wrap` and `Promise.resolve` (Promise). Trials were taken before and after making modifications discussed in Section 10.3.

	Promise	No Promise
Before (ms)	450	250
After (ms)	12	15
Speedup	37.5x	16.6x

Table 10.3: Time taken to take the union of two iterators containing 200,000 elements before, and after making modifications discussed in Section 10.4.

Before (ms)	795
After (ms)	103
Speedup	7.7x

input iterators to be `.read`. The original implementation of the `UnionIterator` was unnecessarily backed by the `BufferedIterator` which means that elements need to be moved into a *linked-list* buffer *prior* to being `.read`.

We modified this behaviour to remove the unnecessary intermediary buffer within this data structure. Instead, of using this buffer - the `UnionIterator` becomes `readable` whenever one of the input iterators becomes readable - and when `.read` is called on the `UnionIterator`, it then forward the call to a `.readable` iterator in its input.

We identified the performance issues in this data-structure[15] and developed the solution[16].

This solution produces approximately an 8x speedup when taking the union of two iterators containing 200,000 elements each. The results of the experiment demonstrating this are given in Figure 10.3.

10.5 Transformations

A large number of the transformation pipelines within the AsyncIterator package consist of repeated `mappping` and `filtering` of data that is being pulled through the iterator pipeline. In the original implementation of the AsyncIterator logic, each `map` and `filter` would result in the creation of a new *buffering* iterator. This resulted in a slowdown of data-processing, exponential with respect to the number of `map` and `filter` operations that were performed[17], due to the fact that each buffer in the 'chain' of `map`/`filter` also

[15] https://github.com/RubenVerborgh/AsyncIterator/issues/52
[16] https://github.com/RubenVerborgh/AsyncIterator/pull/65
[17] https://github.com/RubenVerborgh/AsyncIterator/issues/44

Table 10.4: Time taken to apply an identity map to 2,000,000 elements before, and after making modifications discussed in Section 10.5.

Before (ms)	4736
After (ms)	223
Speedup	21.2x

includes *asynchronous* task scheduling in order to avoid blocking when the buffer is full.

We resolved this performance issue[18] by removing buffers between *synchronous* transformations such as `mapping` and `filtering`. We further optimised this by using transduction (Elliott, 2020) to bypass unnecessary intermediary iterators.

As a consequence of this work, we achieved a 20x performance improvement on the case where there were eight `map` operations[19] applied to an AsyncIterable stream. The results of this experiment are shown in Figure 10.4.

The use of transduction results in a *further* speedup on `filter` operations. This is because, in the original buffered implementation, the iterator would return `null` when an element was filtered out. This `null` result would be passed down the entire chain of iterators, and *then* cause the iterators to be paused and asynchronously rescheduled. Our implementation instead *terminates* calculation at the point where an element is filtered out, *eliminating* subsequent calls down the chain. Furthermore, this implementation is able to *continue* pulling elements from the *source* until an element passes through the chain of transformations without being filtered out, *or* until the source stops producing elements. This *greatly* reduces the amount of *rescheduling* that must take place within the iterator.

10.6 Cloned Iterator

The `CloneIterator` is critical to achieving tree behaviour as discussed in Section 10.1.3. The original iterator, which is `cloned`, stores an array containing each element it emits. Each clone then produces elements by reading items out of the source array. The array storing a history of all elements is, in general, necessary because each of the clones may be reading from a different point in the upstream source.

Several optimisations can be made on common use-cases of this iterator[20].

The first optimisation comes from the observation that a common pattern in the *use* of `.clones` is to perform a series of *transformations* to the clones and then take the `union` of those transformations. This particular case is equivalent to applying a `.multiMap` to the original iterator. Consequently, the typical *memory* requirements of storing an

[18] https://github.com/RubenVerborgh/AsyncIterator/pull/58
[19] https://github.com/RubenVerborgh/AsyncIterator/pull/58
[20] We have not implemented these optimisations as they are beyond the scope of this project.

array of all cloned elements and the *time cost* involved in adding and reading elements from this array can be avoided. This is done by detecting if all of the inputs in a `union` have the same `source` with a series of transformations them, and then using the `MultiTransformIterator` on the source.

We can also avoid the *memory* requirements of instantiating an array in-memory *if* all clones are in *flowing mode*. In this case the clone *source* can control the way in which data is *emitted* from each of the *clones*. In particular, it can choose to emit the item from each of the clones in a *round-robin* fashion. This consequence is that each clone implicitly reads from the *source* in *lockstep* and hence a record of all items is not needed.

Chapter 11

Implementing Native Reasoning Support for N3Store

This chapter principally delivers on **RO2 Performance** by exploring what performance is truly possible when performing RDF reasoning within a browser-client. To achieve this, we develop an RL profile reasoning algorithm that is customised for the triple indexes used within N3Store, an in-memory RDF Store from the N3.js JavaScript package. We shall demonstrate in Part III that this results in a performance that is competitive with many state-of-the-art reasoners.

This work is important because decentralisation efforts such as the Solid protocol have reached a point where they must prove[1] that *fast* RDF Querying and Reasoning is *possible* within the browser. This proof determines the viability of many use-cases within such projects.

Due to the **R01 Interoperable Architecture** developed in Chapter 8, this work seamlessly integrates into the wider T-FIRRE architecture that we have developed. We can define an *actor* that will use the algorithm developed here when one of the input sources to T-FIRRE is an N3Store. This has enabled **RO2 Performance** improvements in certain use cases of our reasoning architecture.

RO3 Accessibility is immediate from the fact that we implement this reasoning algorithm in an RDFJS compliant triple store. In addition we plan to add the RDFJS working `.reason` method we define to the set of standard RDFJS interfaces.

[1] https://github.com/rdfjs/N3.js/pull/295#issuecomment-1125737727

11.1 Overview of N3.js

The N3.js package is the default in-memory data source used by T-FIRRE for storing implicit facts. It is also used throughout Comunica, including use for the caching of data-files in file-based queries. The performance of our work is thus highly dependent upon the performance of this upstream package.

Moreover, this package is extremely popular within the JavaScript ecosystem of the Semantic Web; and powers a large portion of the data management within the JavaScript environment for the Semantic Web. At the time of writing it has 589 stars on GitHub, 336 downstream packages (including Wikidata (Vrandečić and Krötzsch, 2014)) and 2605 dependent GitHub repositories.

Indexing in N3.js In order to understand the following sections, it is worth understanding *how* the N3Store in N3.js stores RDF triples. When a triple is added to the store, the *subject*, *predicate* and *object* are converted to a canonical string (which is the same as the expanded serialisation of that term within a `.n3` document). Each canonical string is mapped to a numerical id. Triples are stored by adding these numerical ids as keys in nested indexes (dictionaries). The N3 Store uses three, three-layered indexes to store data. The depth of these indexes reflects the fact that a triple has three elements subject, predicate and object. The three indexes are ordered as follows:

1. 'object', 'subject', 'predicate'
2. 'subject', 'predicate', 'object'
3. 'predicate', 'object', 'subject'

Each index contains the same information. The three indexes exist to improve the *lookup* time when retrieving data that matches certain patterns.

11.2 Performance Degradation's of Abstract Reasoning Algorithms

The abstract reasoning algorithms that we have implemented in Chapter 8 cannot match algorithms which are optimised for particular storage structures, such as the one that we will discuss in Section 11.3. To understand why this is true in the case of N3.js, we first review the steps that take place when reasoning is applied to the N3Store via the *abstract* reasoning algorithms implemented in T-FIRRE. In particular, lets consider what happens under-the-hood when an *abstract* T-FIRRE rule-evaluation algorithm gets executed with an N3Store is the *source* for explicit data and *destination* for implicit data:

```
match(s, p, o):
    # Convert from RDFJS term to numerical ID
    sID = ids[canonicalise(s)]
    pID = ids[canonicalise(p)]
    oID = ids[canonicalise(o)]

    # Work out the best index to lookup over
    # based on which of sID, pID and oID
    # are defined
    index0 = getIndex(sID, pID, oID)

    # Correctly select the order of term lookups
    # based on which of sID, pID and oID
    # are defined
    key0, key1, key2 = orderLookup(sID, pID, oID)

    # Lookup all of the matches for the pattern given
    # by key0, key1 and key2 over index0 by iterating
    # through relevant parts of index0
    FOR m IN lookup(index0, key0, key1, key2):
        # Convert from numerical ID back to RDFJS term
        subject = canonicalToTerm(terms[m.subject])
        predicate = canonicalToTerm(terms[m.predicate])
        object = canonicalToTerm(terms[m.object])

        # Yield a matching triple via the iterator protocol
        yield triple(subject, predicate, object)
```

Listing 11.1: Algorithm for obtaining the `match`es to triple pattern from an N3Store

1. The premise(s) of the rule are used to generate a Basic Graph Pattern (BGP)[2] template for an RDF Query.
2. Start a *query* to evaluate the bindings[3] of any matches to the BGP template. Which gets evaluated by:
    a) For each pattern within the BGP, retrieve all *matches* from the N3 store. As given in Listing 11.1 this requires the store to:
        i. Convert terms of the *pattern* into internal *id* representatives.
        ii. Look up *matches* to the pattern in the index.
        iii. For each term in a match, look up the *string* representative, and then

[2] https://www.w3.org/TR/rdf-sparql-query/#BasicGraphPatterns
[3] https://www.w3.org/TR/rdf-sparql-query/#select

generate the term *object*.

iv. Generate the matching *quad* from the *terms* in the match.

v. Yield each *quad* using the expensive *iteration protocol*[4].

b) Convert the matching *quad*, to *variable bindings* for the pattern being matched against.

c) Perform an *inner join* (Brumm, 2019) on each stream (Bewig, 2007) of *quad pattern* bindings, to produce a stream of bindings for the BGP patterns.

3. For each element in the bindings stream - generate new quads by substituting the binding patterns into the conclusion of the rule.

4. For each new quad that is generated, *add* it back into the N3 store. To do this internally the store must:

a) Convert each term of the *quad* into a string representation.

b) Look up the numerical *id* of the string representation.

c) Insert the triple with *id* terms onto the index.

In particular, we can see that there is a large amount of overhead involved in doing transformations between *internal indexes*, *quads* and *bindings*.

11.3 Implementing Reasoning on N3.js

Rather than doing this expensive process of converting between numerical ids and RDFJS object representations of terms, we can instead re-write our rules to be in terms of this internal representation and subsequently perform reasoning directly on the indexes of the N3Store. Furthermore, index lookups can be *pre-computed* for each of the rules because we know in advance which terms in the premise will be variables, and which are not. This operation reduces the time complexity of reasoning with respect to the size of the *dataset*.

In particular, the algorithm that we develop[5][6] and formalise in Listing 11.4 runs as follows:

1. Re-write the rules by

a) converting each non-variable term in the premise/conclusion into the internal *id* used by the *N3Store* index;

b) converting each *variable* into a pointer to a memory location into which concrete variables for the variable will be substituted and;

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols
[5] https://github.com/rdfjs/N3.js/pull/295
[6] https://github.com/rdfjs/N3.js/pull/296

c) identifying the dependencies between *conclusions* and *premises* between different rules to identify which rules need to be evaluated next when *new* implicit data is discovered.

Figures 11.1 to 11.9 show the form of these rules in memory and the nature of dependencies between rules. In particular Figure 11.1 conveys how the rule

`?s a ?o ∧ ?o rdfs:subClassOf ?o2 -> ?s a ?o2`

is stored in memory with respect to the mapping in Table 11.1. Figure 11.4 displays the dependencies between the two rules

`?o1 rdfs:subClassOf ?o2 ∧ ?o2 rdfs:subClassOf ?o3 -> ?o1 rdfs:subClassOf ?o3`

and

`?s a ?o ∧ ?o rdfs:subClassOf ?o2 -> ?s a ?o2`

Note the circular references of the first rule to itself are omitted for the sake of diagrammatic clarity.

2. Evaluate the rules by

   a) traversing each term in the premise and looking up all matches directly in the index, for each variable where the pointer is still pointing to an undefined value; substitute the value for a term in the matching quad in the index. This operation is formalised in Listing 11.4 and demonstrated in Figure 11.2 and Figure 11.3.

   b) for each conclusion look up the selected variables for each conclusion and `add` the triple id's to the index

   c) for any *new* triples that are `added` (cf. Listing 11.3) to the store, identify any premises that match the triple, and pre-fill any variable memory locations as shown in Figure 11.6

   d) run reasoning over the resultant rule shown in Figure 11.7 to produce a new conclusion as shown in Figure 11.8 (the global view of what is happening in memory looks like that in Figure 11.9)

   The result is that we use a custom variant of the semi-naive algorithm shown in the `evaluate` function of Listing 11.2 to perform this operation.

```
evaluate():
    buffer ← ∅

    # Perform an initial materialisation
    # of all facts with all rules in the database
    for rule of rules:
        evaluateRule(rule, buffer, 0)

    # Continue reasoning until the buffer of
    # new conclusions is empty
    while buffer != ∅:
        { subject, predicate, object, rule } ← buffer.pop();
        # Prefill the matched premise with the
        # values of the buffered conclusion
        rule.base.subject = subject
        rule.base.predicate = predicate
        rule.base.object = object

        # Apply reasoning on the pre-filled rule
        evaluateRule(rule, buffer, 0);

        # Reset any variable memory components in the rule
        rule.base.reset()
```

Listing 11.2: Evaluating rules over the index of N3.js with a custom variant of the semi-naive algorithm.

```
add(conclusion, buffer)
    # The insert function adds the triple to the
    # internal index.
    # It returns true if the triple was not already
    # stored in the internal index.
    IF insert(
            conclusion.subject.value,
            conclusion.predicate.value,
            conclusion.object.value
        ):
        # If the conclusion was not already in the internal
        # index then add it, any rules that it matches the
        # premise of, to the the buffer for processing
        FOR rule IN next:
            buffer ← buffer ∪ ({
                subject: conclusion.subject.value,
                predicate: conclusion.predicate.value,
                object: conclusion.object.value,
                rule: rule
            });
```

Listing 11.3: `add` function for index-based reasoning on N3.js. This function will `insert` conclusions generated by rules into the store. If the triple did not already exist in the store, it will add the triple and, and any rules with premises matching that triple to a buffer to be evaluated.

```
evaluateRule(rule, cb, i, buffer):
    premise ← rule.premise[i].premise
    # Get the correct index (dictionary) to search over
    index ← rule.premise[i].index
    # [a] Get the key that we wish to look up in
    # the first layer of the index.
    value = premise[0].value

    # [b] If a key is specified, use only that part of index
    # otherwise, we are interested in all values
    if value != undefined:
        index ← { [value]: index[value] }
    FOR value IN index:
        index ← index[value] # [c] Lookup nested index
        premise[0].value ← value # [d] Set mem. variable

        value ← premise[1].value # See [a]
        IF value != undefined: # See [b]
            index ← { [value]: index[value] }
        FOR value IN index:
            index ← index[value] # See [c]
            premise[1].value ← value # See [d]

            value ← premise[2].value # See [a]
            IF value != undefined: # See [b]
                index ← { [value]: index[value] }
            FOR value IN index:
                premise[2].value ← value # See [d]
                IF i < rule.premise.length - 1:
                    # If not the last premise in rule,
                    # index over the next premise
                    evaluateRule(rule, i, buffer)
                ELSE
                    # If this is the last premise, all
                    # variable memory locations are now
                    # filled. Conclusions are materialised.
                    FOR conclusion IN rule.conclusion:
                        add(conclusion, buffer)

                # Clear any variable in-memory locations
                premise[2].reset()
            premise[1].reset()
        premise[0].reset()
```

Listing 11.4: `evaluateRule` function for evaluating a single rule over N3.js. This function iterates through each of the premises to fill in variable memory locations. Once all variable memory locations are filled the conclusions are materialised.

Table 11.1: Mapping between integer IDs (left) and IRIs (right)

1	rdf:type
2	rdfs:subClassOf
3	timbl:me
4	ex:computerScientist
5	foaf:Person
6	ex:computerScienceProfessor
7	ex:Armin

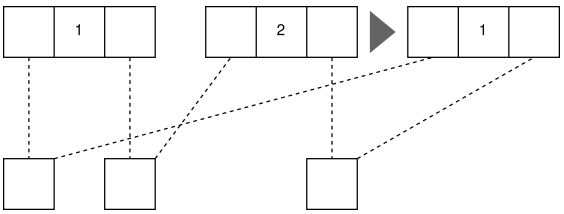

Figure 11.1: Representing the rule `?s a ?o` ∧ `?o rdfs:subClassOf ?o2 -> ?s a ?o2` in N3.js. 1 is `rdf:type` and 2 is `rdfs:subClassOf` in accordance with Table 11.1.

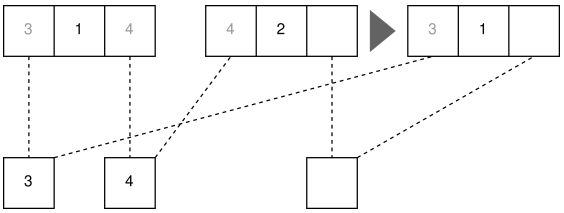

Figure 11.2: The rule from Figure 11.1 with the first premise matched with the triple `(timbl:me, a, ex:computerScientist)`. Hence 3 is `timbl:me` and 4 is `ex:computerScientist` in accordance with Table 11.1.

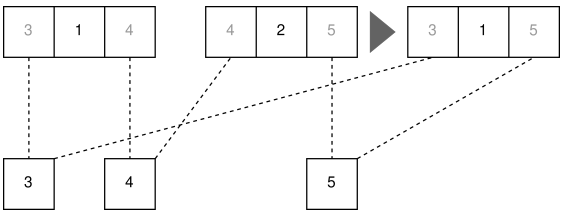

Figure 11.3: Matching the second premise of the partially filled rule in Figure 11.2 against the triple `(ex:computerScientist, rdfs:subClassOf, foaf:Person)`. Hence 5 is `foaf:Person` in accordance with Table 11.1.

Similar to how we observed that advances in server side reasoning *complement* the work that we are doing, it is also the case that advances in *single source* reasoning for *browsers*

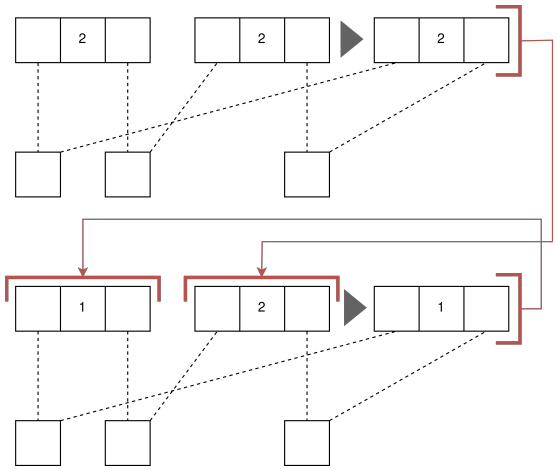

Figure 11.4: Representing the rule `?o1 rdfs:subClassOf ?o2` ∧ `?o2 rdfs:subClassOf ?o3 -> ?o1 rdfs:subClassOf ?o3` (top) and `?s a ?o` ∧ `?o rdfs:subClassOf ?o2 -> ?s a ?o2` (bottom) in N3.js in addition to the dependencies between rules. Here 1 is `rdf:type` and 2 is `rdfs:subClassOf` in accordance with Table 11.1.

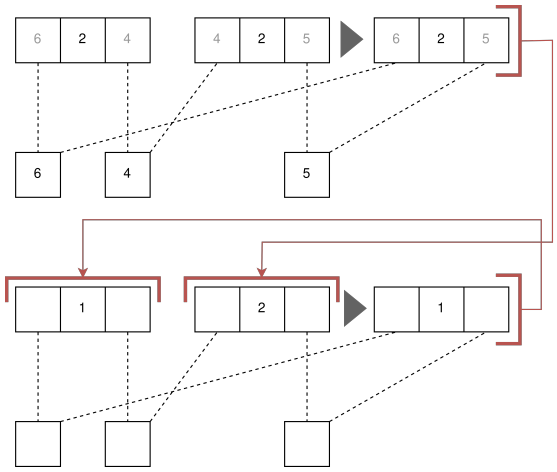

Figure 11.5: The rules from Figure 11.4 with the the premises in the first rule now matched against `(ex:computerScienceProfessor, rdfs:subClassOf, ex:computerScientist)` and `(ex:computerScientist, rdfs:subClassOf, foaf:Person)`. Hence 4 is `ex:computerScientist`, 5 is `foaf:Person` and 6 is `ex:computerScienceProfessor` in accordance with Table 11.1.

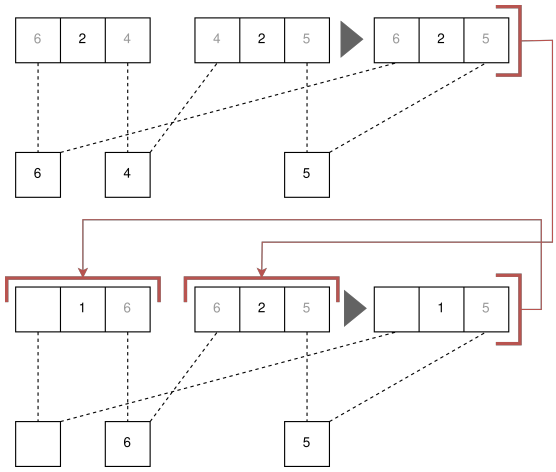

Figure 11.6: Using the conclusion generated in Figure 11.5 to pre-fill a premise in the second rule. Hence 4 is `ex:computerScientist`, 5 is `foaf:Person` and 6 is `ex:computerScienceProfessor` in accordance with Table 11.1.

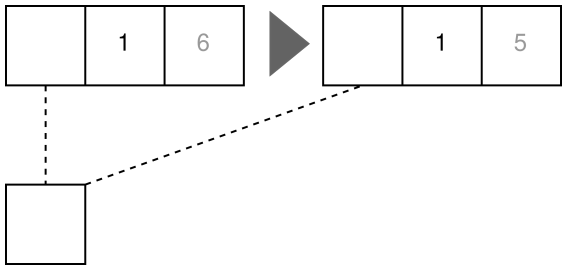

Figure 11.7: The rule that needs to be evaluated by the reasoner after the pre-filling that has taken place in Figure 11.6. Hence 5 is `foaf:Person` and 6 is `ex:computerScienceProfessor` in accordance with Table 11.1.

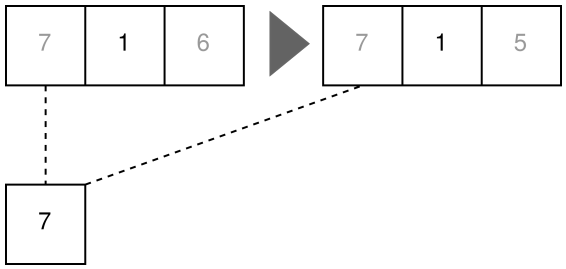

Figure 11.8: Matching against the triple `(ex:Armin, a, ex:computerScientistProfessor)` in the rule in Figure 11.7 to produce the triple `(ex:computerScienceProfessor, a, foaf:Person)`. Hence 5 is `foaf:Person`, 6 is `ex:computerScienceProfessor` and 7 is `ex:Armin` in accordance with Table 11.1.

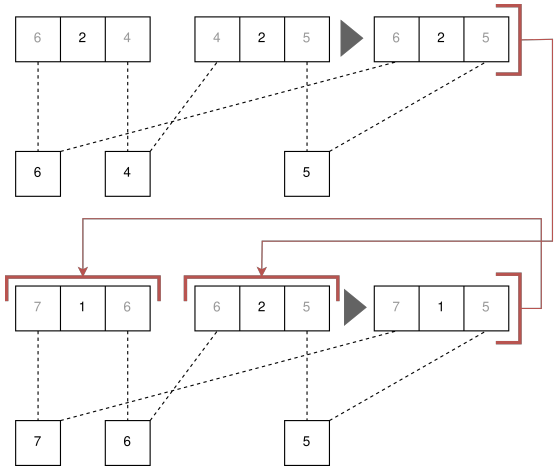

Figure 11.9: Global view of the result of the match made in Figure 11.8. Hence 4 is `ex:computerScientist`, 5 is `foaf:Person`, 6 is `ex:computerScienceProfessor` and 7 is `ex:Armin` in accordance with Table 11.1.

also complements our work. This is because, in cases where there is a single in-memory source *with reasoning support* the T-FIRRE architecture that we have developed can automatically run reasoning at that *lower abstraction layer* to achieve an improved performance in comparison to the *abstract* reasoning algorithm implemented using T-FIRRE internals that is capable of reasoning over *any* data-source. The consequence of this, is that this work on reasoning over *internal indexes* enables *increased performance* over inferences made on data that exists *in-memory* in the browser; whilst our work on T-FIRRE enables the *capability* of performing federated reasoning across in-memory and remote sources in a *network-efficient* manner.

11.4 Optimising Existing Functionality of N3.js

Given the critical nature of the N3.js library across many projects, including Comunica and T-FIRRE - we thought it valuable to spend time producing *performance* optimisations for this library.

11.4.1 The `match` method in all cases

The core primitive that T-FIRRE and Comunica use from this library, is the `match` method. This method accepts the pattern of data to be extracted from the store - for instance, one could request all triples that match the pattern `(timbl, a, _)`. This method then returns an iterable stream of data.

Prior to our work on the library, the time taken to `match` against 16.8M triples was 41.5s. To achieve this operation the library would:

1. Materialise all of the matches to be made within an array[7].
2. Load those matches into a new N3Store[8].
3. Load those quads back into another array[9].
4. Iterate over that array.

We were able to reduce the time taken to iterate over all of these 16.8M matches to 3.26s - this was achieved by enabling asynchronous iteration over the indexes[10]. These improvements were included in version 1.16 of N3.js, and our results are shown in Figure 11.10.

We also provided an additional optimisation of removing unnecessary repeated calculations of terms from id representations. This improvement over version 1.15 and 1.16 of N3.js is also shown in Figure 11.10.

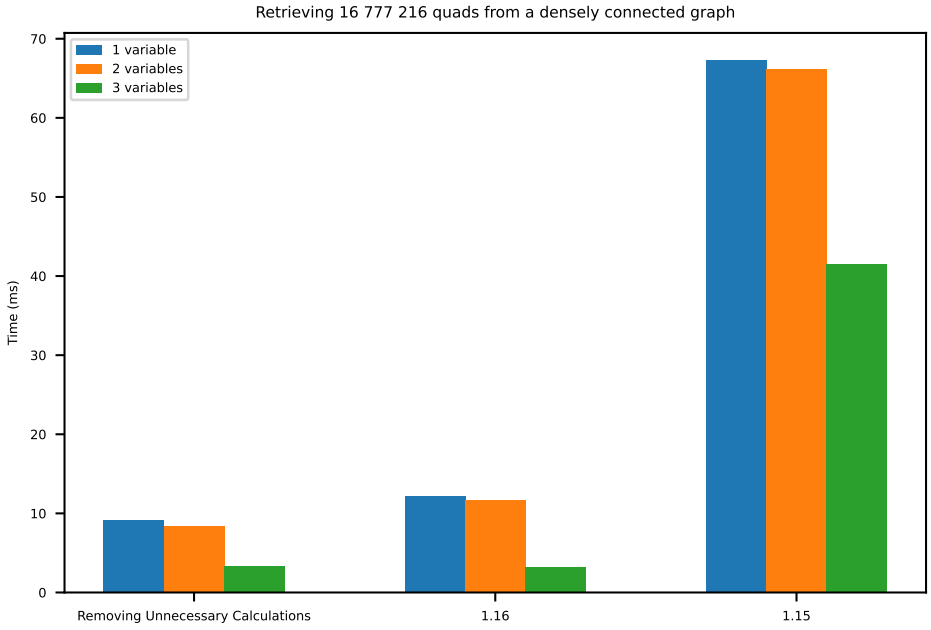

Figure 11.10: Performance of N3 data reading

[7] https://github.com/rdfjs/N3.js/blob/cf8d28f0c05fc327151c1a76e333f2f8f0fce819/src/N3Store.js#L844
[8] https://github.com/rdfjs/N3.js/blob/cf8d28f0c05fc327151c1a76e333f2f8f0fce819/src/N3Store.js#L845
[9] https://github.com/rdfjs/N3.js/blob/cf8d28f0c05fc327151c1a76e333f2f8f0fce819/src/N3Store.js#L873
[10] https://github.com/rdfjs/N3.js/pull/275

11.4.2 The `match` method on sparse graphs

A second core issue in this library is that it failed to efficiently handle `matching` against KGs with sparsely connected entities, in cases where there was only one non-variable term in the triple pattern being matched against (for instance `(_, a, _, _)`). This was due to the code designed to optimise the *all-variable* case resulting in a slowdown due to also trying to handle the one non-variable case. We fixed this performance issue and the improvements are shown in Figure 11.11.

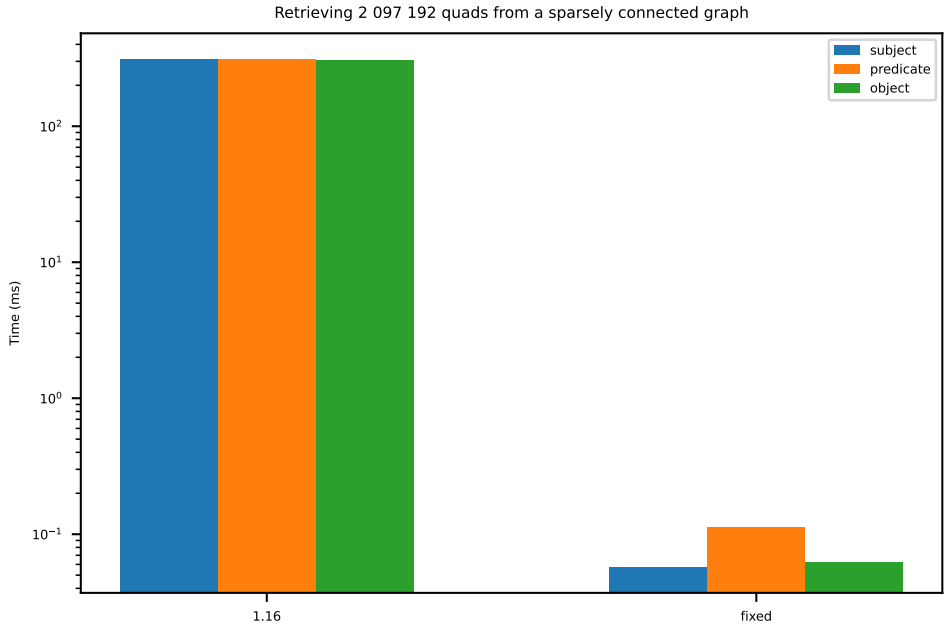

Figure 11.11: Performance of N3 data reading on sparse data-sets

11.4.3 Future performance improvements

Despite our work, there is still a significant amount of work that can be done to further improve the performance of the library.

Skipping Quad generation when loading data into a store. Another place where unnecessary conversions are made is in loading data into an RDFJS store from a string source (such as a local file). This is another common operation when reasoning with T-FIRRE due to the way that documents such as local files are reasoned over. Typically the flow for this operation is that data is parsed into RDFJS Quads, and then these Quads are added to the store where they are converted to string representations, and in turn, internal IDs. In most cases the internal string representation used in the store,

is the same as the string representation in the document, and hence the intermediary RDFJS representation is unnecessary. Hence we propose an extended RDFJS store interface which includes a `load` method which can load in various *streams* of strings that are parsed and loaded into the database without going through unnecessary internal representations.

Shared Entity Index. In order to prevent the repeated conversion between internal *ids* and full RDFJS quad representations (for instance in applications such as T-FIRRE and Comunica), it would be useful to have a shared `EntityIndex`[11] object that can be referred to by several stores, T-FIRRE and Comunica. This would enable multiple stores to share the same index between numerical and string ids, thus preventing unnecessary memory overhead. Furthermore this index could also be shared with the likes of the entity record introduced in Section 9.4.

Lazy generation of Quads. When Quads are generated in Comunica all terms that are matched against are eagerly converted from internal numerical representations into RDFJS terms for the *subject*, *predicate*, *object* and *graph*. However in many use-cases only a subset of such terms are read. For instance in the SPARQL query `SELECT ?s WHERE ?s ?p ?o` only one of the four terms is read once it has left the store. In turn this means that RDFJS representations for the predicate, object and graph never need to be generated. To avoid this, the Quad can instead just hold the internal IDs and a reference to the shared Entity Index. Each of the terms in the quad can then become getter[12] rather than strict attributes which lazily generate the required RDFJS term on-demand.

`matchPattern` method. In the case of patterns like `(?v, a, ?v)` these can be looked up much faster than matching against the blanks `(_, a, _)` by linking the keys within `_findInIndex`.

We propose a `matchPattern` method be added to the RDFJS data spec which is able to handle repeated variables of this nature when searching through the index.

More importantly - to enable much faster `SELECT DISTINCT` queries and reasoning over data sources it would be useful to either have as part of `matchPattern`; or even `match` itself, to specify that we only care about particular entries in a pattern. For instance running `match(false, null, false)` would essentially be a way of listing out all of the predicates in the graph. This can significantly reduce the number of quads that applications like T-FIRRE and Comunica would have to iterate over in some situations.

[11] https://github.com/rdfjs/N3.js/issues/289#issue-1205977629
[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get

Part III

Epilogue

Chapter 12

Evaluation

This chapter evaluates to what extent we have enabled *interoperable* and *performant* query enrichment on a decentralised Semantic Web. In pursuing our research objective, **R02 Performance**, we sought to demonstrate that it is *possible* to perform reasoning in the browser in a manner that is *performant enough* for *standard use-cases* to execute without interrupting the user experience in applications that make use of the reasoner. We further sought to investigate what reasoning performance is *truly possible* within the browser.

Accordingly, we evaluate the *abstract* reasoning algorithms that we created as part of our T-FIRRE reasoning architecture in Chapter 8. The performance of these algorithms is further improved through the data structure optimisations in Chapter 10 and the theoretical developments in Chapter 9 that are implemented as *optimisation actors*.

Furthermore, we evaluate the performance of the index based reasoner developed in Chapter 11 to examine the reasoning performance that is *truly possible* within the client.

To assess the performance of our reasoners in comparison to other state-of-the-art reasoners we shall apply benchmarks that are widely used within the Semantic Reasoning Community.

In order to evaluate our **R02 Performance** objective, we shall consider the use-case of performing federated reasoning across the union of a personal profile document, and the FOAF ontology. It is widely agreed that 0.1s is the maximum delay permissible within a system to enable it to seem as though it is reacting instantaneously to users (Nah, 2004). Hence, the aim of our evaluation is to perform reasoning on this use-case in $<\sim$ 0.1ms.

We note that our primary objective is not just one of absolute performance. Rather, our work seeks to enable *interoperable*, *federated* reasoning on the client-side. For this reason, we cannot expect to outperform reasoning implementations, such as RDFox, that are far less configurable and not portable to a browser environment due to the language

that it is implemented in, and the fact that the package is too large to be included in Web applications.

12.1 Overview of Reasoners to be Evaluated

In this chapter, we perform an evaluation of several of the RL reasoners that we have developed over the source of the project. We evaluate the following reasoners that we have developed:

- ***n3*** - Our reasoning engine for N3.js[1] that was introduced in Chapter 11. We shall show reasoner has a *highly competitive performance* even compared to *state-of-the-art* reasoners. As discussed in Chapter 11, this reasoner uses a modified version of the *semi-naive* algorithm that pre-computes index lookup.
- ***com. restrict*** - Our T-FIRRE implementation of the *rule restriction* algorithm. The implementation follows the architecture given in Figure 8.3 from Chapter 8.
- ***com. forward*** - Our T-FIRRE implementation of the *semi-naive forward chaining* algorithm. This implementation follows the architecture of Figure 8.5 from Chapter 8 but uses generic `AsyncIterator` methods instead of defining custom `AsyncIterator` extensions.
- ***com. direct*** - The fastest abstract reasoning algorithm we have implemented for our reasoning engine in T-FIRRE. This algorithm is a variant of the *semi-naive forward chaining* shown in Figure 8.5 from Chapter 8. Internally, this reasoner makes use of custom extensions of the `AsyncIterator` data structure to optimise certain operations that are performed.

We compare these reasoning algorithms that we have developed against the state-of-the-art open-source reasoners. We also benchmark our work against the HyLAR engine (Terdjimi et al., 2015) which, as discussed in Chapter 3, was the only browser-compatible reasoner prior to the start of our work.

- **CWM**[2] - A general-purpose data process for the Semantic Web. CWM is a forward chaining reasoner written in Python.
- **EYE** (Verborgh and De Roo, 2015) - A Prolog reasoning engine supporting the Semantic Web layers. The engine performs semibackward reasoning in addition to supporting Euler paths.
- **HermiT** (Glimm et al., 2014) - An OWL2 compliant RDF reasoner developed by the creators of RDFox. The reasoner is based on the *hypertableau calculus*.
- **jDREW** (Craig, 2007) - An 'Object Oriented java Deductive Reasoning Engine for the Web'.

[1] https://github.com/rdfjs/N3.js/pull/295
[2] https://www.w3.org/2000/10/swap/doc/cwm.html

- **Jena**[3] - The Java-based Apache Jena inference engine.
- **HyLAR** (Terdjimi et al., 2015) - The only other browser-compatible reasoner prior to our work.

12.2 Experimental Setup

The experiments were performed on a DELL XPS 15 9500 laptop with 32GB of RAM using Node v18.1.0. We note that NodeJS sets the default memory limit to 4GB[4]; and that we use this default in all experiments, *except* when performing the Deep Taxonomy Benchmark at a depth of $1,000,000$, in which case we increase the memory limit to 8GB to be able to handle the requisite amount of data.

The V8 Engine The V8 engine[5] is the high-performance JavaScript engine backing NodeJS. The V8 engine is built for and used by Google Chrome. The V8 engine is written in C++ and largely achieves its high performance by identifying common patterns in code/data-structures and performing the associated operation in C++.

Consequently, the performance we observe in our JavaScript reasoner implementations can partly be attributed to the optimisations made in this engine - and would not have been possible in previous JavaScript engines.

While other browsers use different JavaScript engines, for example, Firefox uses the SpiderMonkey engine, these engines are also receiving large amounts of work in order to compete[6] (Parravicini and Mueller, 2021) with the performance that is seen with V8, and in some cases, outperform V8[7]. For this reason, we consider our evaluations, which are performed on the NodeJS V8 engine, to be a strong indicator that it is *viable* to perform client-side reasoning across most browsers.

12.3 The Deep Taxonomy Benchmark

The Deep Taxonomy Benchmark[8] is designed to evaluate the performance of reasoners on *deeply nested* subClass hierarchies.

Reasoners that implement a *naive* forward-chaining approach, tend to perform poorly on this benchmark, as the *nesting* of classes results in such reasoners *repeatedly* re-evaluating facts throughout the reasoning process.

We evaluate the performance of our reasoners against those that have been previously

[3] https://jena.apache.org/documentation/inference/
[4] https://medium.com/geekculture/node-js-default-memory-settings-3c0fe8a9ba1
[5] https://v8.dev/
[6] https://arewefastyet.com/linux64/benchmarks/overview?numDays=60
[7] https://bugs.chromium.org/p/v8/issues/detail?id=12730
[8] http://eulersharp.sourceforge.net/2003/03swap/dtb-note

Table 12.1: Time (ms) for the reasoners we developed to apply materialisation the the Deep Taxonomy Benchmark

Depth	restrict	fwd	direct	n3
10	0.024	0.008	0.001	0.001
100	0.667	0.083	0.005	0.003
1000	57.411	0.443	0.057	0.026
10000		3.704	0.231	0.114
100000		40.861	2.034	0.676
1000000			27.694	7.854

evaluated on this benchmark. Our performance for these experiments is given in Figure 12.1. The corresponding raw data is given in Table 12.1 and Table 12.2.

The ***com. direct*** and ***n3*** reasoners that we have implemented outperform every other reasoner that is benchmarked across all trials. This is a very promising indicator that we have successfully created reasoners with a *state-of-the-art* performance, and thus indicates that we have likely achieved **R02 Performance**. Furthermore, we observe that the ***com. direct*** and ***n3*** reasoners that we have developed comprise 2 of the 4 reasoners that are capable of reasoning at a depth of $1,000,000$. Both of these reasoners outperform the Prolog EYE reasoner and CWM reasoner at this depth. This is despite the fact that the EYE reasoner is tuned to perform well on the Deep Taxonomy Benchmark and advertises its' performance on the benchmark[9]. However, the speed that we have achieved with the ***com. direct*** reasoner still cannot match that of the reasoner that we have implemented over the N3.js store. This reasoner we implemented on the N3.js store is capable of evaluating the benchmark in 7.8ms which is more than 3.5x faster than the 27.69ms required by the ***com. direct*** reasoner. The ***com. fwd*** and ***com. direct*** algorithms cannot reach a depth of $1,000,000$ without exceeding the memory limits of the experiment. This is because these implementations using generic `AsyncIterator` methods are forced produce an an excessive number of `AsyncIterator` objects to handle the dynamic nested joins. These objects consume the memory.

We note that the HyLAR engine is outperformed by the results achieved by the *client-side* reasoners that we have developed. We also note that the HyLAR reasoner has a reasonably good performance at shallow depths for reasoning but it does not scale well at greater depths, as a result of making use of more *naive* reasoning algorithms for materialisation.

[9] http://eulersharp.sourceforge.net/#benchmarks

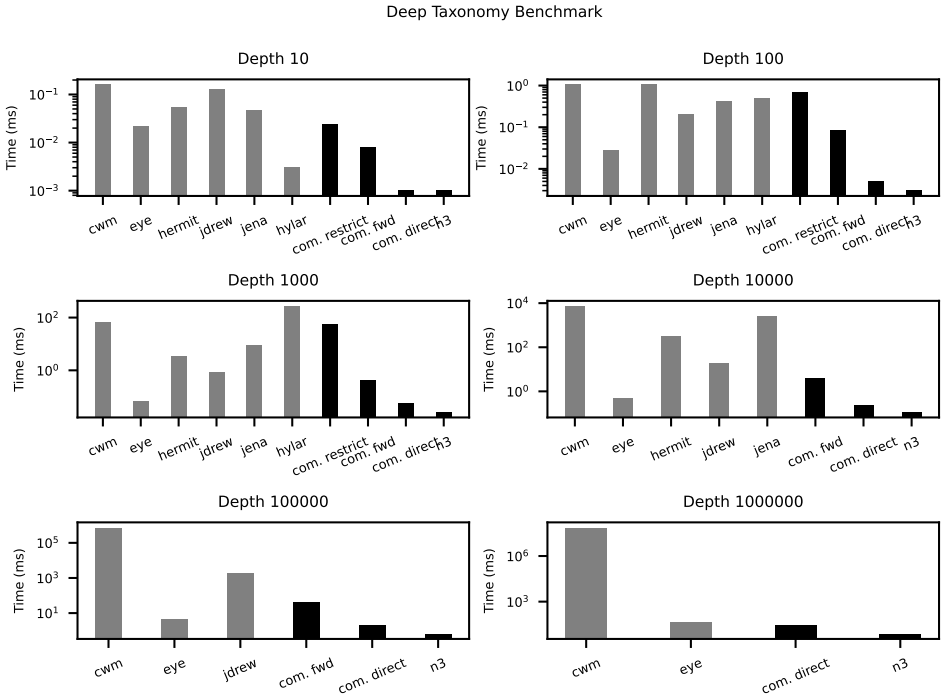

Figure 12.1: Performance of reasoner implementations on the deep taxonomy benchmark. The omission of a reasoner from a diagram indicates that it exceeded memory requirements before reasoning terminated.

Table 12.2: Time (ms) for existing reasoners to apply materialisation the the Deep Taxonomy Benchmark

Depth	cwm	eye	hermit	jdrew	jena	hylar
10	0.16	0.022	0.055	0.13	0.047	0.003
100	1.05	0.028	1.04	0.2	0.422	0.496
1000	65.93	0.066	3.58	0.87	9.302	270.558
10000	7298.0	0.484	310.51	18.68	2597.242	
100000	732974.07	4.617	1875.0			
1000000	73267200	46.648				

12.4 The LUBM Benchmark

The Lehigh University Benchmark (LUBM) (Guo et al., 2005) was 'developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way'[10]. The benchmark uses synthetic data - and is targeted at testing RL profile existential queries over large data-sets.

The performance of the reasoners that we have developed is given in Figure 12.2. As a point of comparison we give the time taken to *load* the data from the LUBM benchmark into an N3.js store.

These results show that the reasoning times with ***N3*** and ***com. direct*** are faster than the time taken to *load* the data into the dataset. This again indicates that we have achieved **R02 Performance** as the time taken for *reasoning* is negligible in comparison to other operations within the data-pipeline.

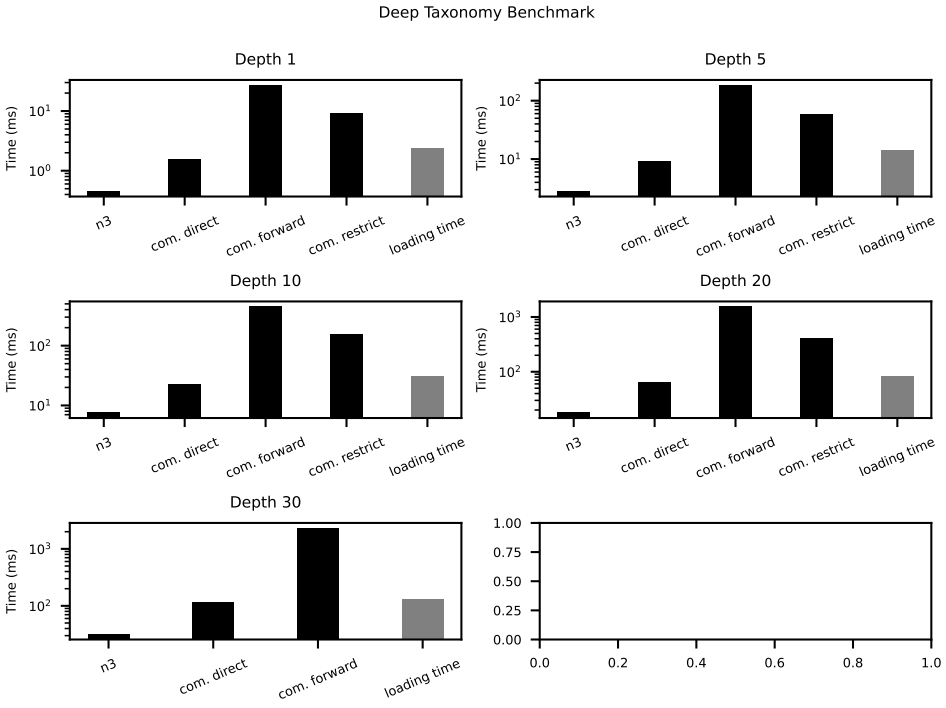

Figure 12.2: Performance of reasoner implementations on the LUBM benchmark. The omission of a reasoner from a diagram indicates that it exceeded memory requirements before reasoning terminated.

[10] http://swat.cse.lehigh.edu/projects/lubm/

12.5 Reasoning Over a Profile Card and the FOAF Ontology

Finally, we evaluate the performance of our implementation over a common use-case within the SOLID ecosystem, which is reasoning against the union of a personal profile card from a SOLID Pod[11] and the OWL 2 (Hitzler et al., 2009) FOAF ontology (Brickley and Miller, 2007).

Our example reasons over the personal profile card of *Tim Berners-Lee*[12] and the FOAF ontology[13].

The ability to perform reasoning on this data in a time frame that is *negligible to humans* demonstrates that, with our reasoners, we have achieved the performance required for them to be included within user-applications.

The purpose of this particular benchmark is to evaluate the *viability* of our work in a production environment, rather than how it *competes* with other implementations of reasoners. As such, we only analyse the performance of the reasoners that we have implemented.

Our results shown in Figure 12.3 indicate that we achieve this desired performance. In particular, the N3Store reasoner is capable of applying reasoning to this dataset in just 43ms and our *abstract* reasoning algorithms can all apply reasoning in less than half a second. Our fastest T-FIRRE configuration (***com. direct***) is able to apply reasoning to this dataset in 108ms.

The consequence of these results is that we are able to apply reasoning from Comunica in a time frame which could appear to users as though the results are being produced instantaneously (Nah, 2004).

12.6 Discussion of Results

There are several impactful consequences of the results that we see in this evaluation.

Importantly we note that the abstract ***com. direct*** reasoning algorithm that we have implemented in our reasoning architecture achieves **R02 Performance** by making it '*possible* to perform reasoning in the browser in a manner that is *performant enough* for standard *use-cases*'. Our index-based N3.js reasoner then achieves the stretch goal of **R02 Performance**, demonstrating that an even greater performance is possible to the extent that it is feasible compete with state-of-the-art reasoners from the browser client. In some cases this implementation can even produce inferences on benchmarks where other reasoners fail with memory errors. This provides the first set of *evidence* demonstrating that it is possible to efficiently perform a wide range of real-world RDF

[11] https://solidproject.org/
[12] https://timbl.inrupt.net/profile/card
[13] http://xmlns.com/foaf/spec/

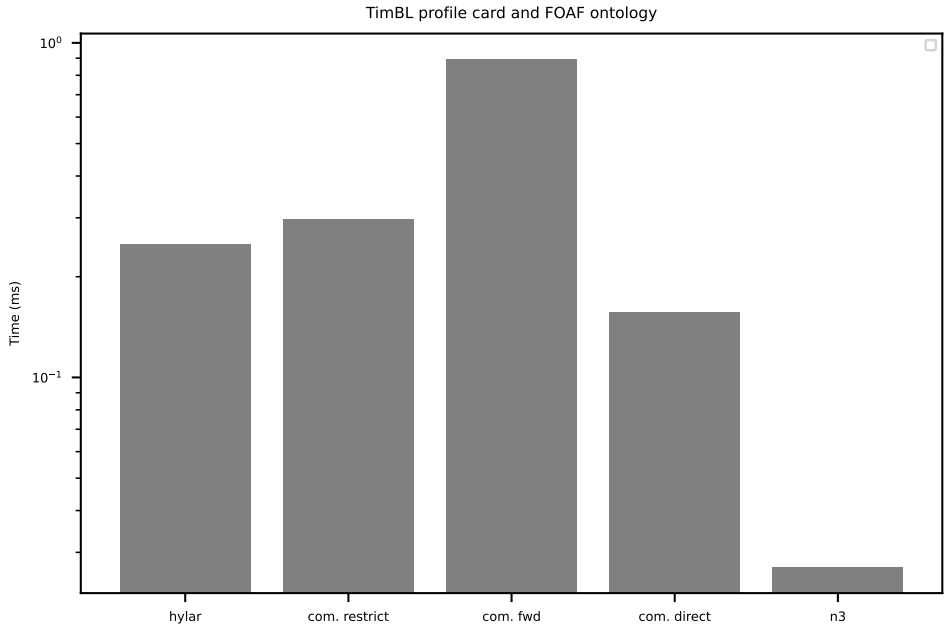

Figure 12.3: Performance of reasoner implementations applying the RDFS rule set to the union of the FOAF ontology and *Tim Berners-Lee*s personal profile card.

reasoning tasks from the *browser-client*[14]. Additionally, we recall that the *interoperable* reasoning components that we have developed for T-FIRRE are capable of delegating the task of reasoning to the lowest possible abstraction layer, supported by **R01 Interoperable Architecture**. This means that in use-cases where the T-FIRRE reasoning bus is tasked with reasoning over a single in-memory N3.js store, it can delegate the task to an actor that simply invokes the `.reason` method on N3.js.

Our best *abstract* reasoning algorithm for T-FIRRE, whilst not as fast as reasoning directly over the indexes in N3.js, is still competitive with the performance of many state-of-the-art reasoners. We observe that each of our abstract reasoning algorithms are capable of reasoning against arbitrary sources and destinations and thus provide the important functionalities of federation and interoperability of reasoning. Hence, this provides evidence that it is possible to efficiently executed many real-world RDF reasoning tasks, that require *federation* across multiple sources, using the architecture that we have developed in our paper.

Furthermore, we hypothesise[15] that it is possible to further improve the performance of our reasoning components in T-FIRRE to achieve comparable performance to the direct implementation on N3.js. This will be achieved by:

- Using ID representation of quads and terms, rather than actual values[16] (where possible) to remove the overhead of converting between term and ID, when retrieving and storing data; and
- Reducing the number of *iterators* that are created and destroyed throughout the current implementation of reasoning within the Comunica internals.

Overall, we effectively completed **R02 Performance** by clearly demonstrating that it is possible to achieve client-side reasoning in a performant manner. Further we have shown that this research objective is satisfied by our interoperable T-FIRRE reasoning architecture, and extended by our index based reasoning on N3.js.

[14] https://github.com/rdfjs/N3.js/pull/295#issuecomment-1125737727
[15] https://github.com/rdfjs/N3.js/pull/295#issuecomment-1123056587
[16] https://github.com/rdfjs/N3.js/issues/289#issue-1205977629

Chapter 13

Conclusion

The work of this thesis makes a significant contribution in the domain of Semantic Web reasoning. In particular, we develop the first *interoperable* and *performant* browser-compatible RDF reasoners that are capable of executing many real-world applications of RDF reasoning. The outcomes that we have achieved in this work are particularly impactful in emergent extensions of the Semantic Web such as the SOLID protocol. In SOLID, work is underway to create an interoperable data architecture in which applications can enforce assumptions around the nature of the data. Our work in this thesis makes this possible for data reasoning in the browser.

The outcomes were achieved through the development of an RDFJS compliant architecture, which can execute various reasoning algorithms based on the sources that are to be reasoned over, in addition to metadata available about these sources. These algorithms include abstract reasoning algorithms that handle any set of sources and destinations; and source-specific algorithms such as the *high-performance* N3.js index reasoner.

13.1 Research Objectives

This thesis has met and exceeded each of the research objectives defined in Section 1.2. In the following, we highlight our achievements in relation to each of the research objectives.

13.1.1 R01 Interoperable Architecture

We have developed T-FIRRE, an *interoperable* and *federated* RDF reasoning engine that integrates with the Comunica Framework. The T-FIRRE engine which we have created, and described in Chapter 8 enables users to define the *rules* and *explicit data sources* that they wish to apply reasoning to. Our architecture will guarantee that appropriate reasoning algorithms are applied to ensure that the correct *implicit facts* are included in the results. Moreover, the T-FIRRE architecture we have created is designed so that

reasoning can be *interoperably* applied at different layers of abstraction depending on the types of data sources that reasoning is being applied to; and what pre-reasoning has been applied to those sources.

13.1.2 R02 Performance

We have developed several *performant* reasoners throughout our work. Firstly, we created performant *abstract* reasoning algorithms in the T-FIRRE engine. In Chapter 11 we then developed a 'lighting fast' RL profile reasoner on the indexes of the N3Store from the N3.js library. This N3Store reasoner can also be used to improve reasoning performance within the T-FIRRE Reasoning Engine that we have developed due to the **(R01) Interoperable Architecture** we developed. The evaluation in Chapter 12 demonstrates that the performance of the latter N3Store reasoner that we developed is competitive with several *state-of-the-art* reasoners. Importantly, our results prove that it is *possible* to perform reasoning in the browser in a manner that is *performant enough* for *standard use-cases* to execute without interrupting the user experience in applications that make use of the reasoner.

13.1.3 R03 Accessibility

Our architecture, by design, is compliant with RDFJS standards and is compatible with many existing libraries and applications within the RDF JavaScript ecosystem. Due to the RDFJS compliance of our T-FIRRE reasoner, and its integration in the Comunica Framework, it can be used in client side tooling like LDflex[1] off the shelf. This integration with the remainder of the RDFJS ecosystem means that front-end developers can now for the first time, easily access and control reasoning behaviour.

As an example of the *straightforward* experience that this offers developers, consider the code snippet in Listing 13.1. Developers have access to enriched data *via* a quickstart version of the LDflex (Verborgh and Taelman, 2020) API. We can see that there are 8 boilerplate lines of code to initialise the wrapped engine; and then one line to ask the question 'Is Tim Berners-Lee a Human?'.

[1] https://github.com/ldflex/ldflex

```
import { createPath } from "@ldflex/comunica-quickstart";
const timbl = createPath(
  "https://timbl.inrupt.net/profile/card#me", {
  reasoning: true,
  traversal: true,
  context
 }
)

const isHuman = await timbl.type.ask("foaf:Person");
```

Listing 13.1: Querying over enriched data using the LDflex API

Under the hood of this last line of code, the T-FIRRE reasoner that we have developed will perform *entity alignment* with any of the other definitions of *Tim Berners-Lee* that are the `owl:sameAs` https://timbl.inrupt.net/profile/card#me. Following this, reasoning will then be used to resolve the `rdfs:subClassOf` hierarchies to see if *Tim Berners-Lee* is a `foaf:human`, and finally any relevant quads will be returned from our architecture, and passed to the Comunica Query Engine. In this case, we are performing a *ask* query, so the query engine will then return `true` if there are any matches and `false` otherwise.

Furthermore, all of our work towards enabling *interoperability*, discussed earlier, is relevant. The reasoner we have developed is capable of delegating reasoning tasks to as lower abstraction layer as possible (depending on the metadata published by servers, documents and in-memory stores); this enables a result to be returned *as quickly as possible.*

13.1.4 R04 Future-Proof Design

Our work developing T-FIRRE provides the *first viable* end-to-end solution that enables developers to specify that they would like to have reasoning applied to the data in their application, without the developers needing to have an understanding of the internals required for this to happen. Moreover, we have crafted our work so that this API used by developers will remain stable as future work and developments take place. The modular design of T-FIRRE is extensible, enabling the addition of new reasoning actors that can improve the performance and expressivity of federated reasoning. Such possible extensions include:

- Regional aggregators (which aggregates and *reasons* over data from a set of databases in a server), to reduce the number of endpoints that the client has to federate over.
- Dialogical reasoning via WebSockets (getting a server to handle a large amount of the processing required for reasoning).
- WASM reasoners, which hope to bring faster reasoning to the browser through

the access to lower-level constructs, and better parallelism than that provided by WebSockets. However, this appears to be difficult with the current WASM 1.0 standard that is implemented due to the high cost of *message passing* when trying to parallelise tasks.

Moreover, our design enables reasoning to be applied to, and experimented with in conjunction with other research directions via dependency injection. We will discuss these possibilities further in Section 14.

Chapter 14

Future Work

In the following sections we discuss future research directions following this work. We discuss the notion of future work from several angles. We start with a broader analysis of the interoperability work that is still required within SOLID and the Semantic Web in general. We then narrow down to discuss specific next steps for the T-FIRRE and N3Store reasoners that we have built in this paper.

14.1 Interoperability in SOLID

The SOLID decentralised protocol has seen much recent activity/interest. Inrupt, *Tim Berners-Lee*'s private company developed to develop the protocol, has received 50M USD worth of investments[1] and the Flemish Government has allocated 7M€ funding[2] for a SolidLab at the University of Ghent.

The underlying technical challenge that must be fulfilled in order to ensure the success of this protocol, is that of *data interoperability*. In order for applications to flourish in such an ecosystem, there must be clean, interoperable interfaces that they can use for *discovery*, *authentication*, *collection*, *enrichment*, *validation*, *query* and *update*s to data.

In this work, we have focused on the challenge of interoperable *enrichment*. The challenges of interoperable *authentication*, *collection*, and *query* have largely been solved by the existing architecture of the Comunica Engine[3], and its integration with Inrupt's

[1] https://www.crunchbase.com/organization/inrupt/company_financials
[2] https://www.ugent.be/ea/idlab/en/news-events/news/flemish-government-7m-eur-funding-solidlab-vlaanderen.htm
[3] https://github.com/comunica/comunica

authentication SDKs[4][5]. That of *discovery*[6] and *update*[7] are under active research within the engine. The primary remaining requirement is that of *validation*[8], which has seen less active development in an *interoperable* context. In the near future, interoperable tools like the Comunica should start researching how to effectively introduce validation into their pipeline, so as to allow developers to ensure that requisite *validations* are made to the underlying data. This validation should be done using existing standards like SHACL Knublauch and Kontokostas (2017) and ShEx Prud'hommeaux et al. (2019); and in the same way we have done with reasoning, have a fallback validator in the browser, but delegate the validation process to lower abstraction layers; including in-memory stores, or remote servers where possible. This enables app developers to ensure that any assumptions they make about the *explicit data* can be verified before running their app.

Further to this - there is orthogonal research into *data discovery* that is required to work out how to intelligently select subsets of a knowledge graph to apply reasoning to; without having to perform backward *or* forward chaining. For this direction of research we propose the development of a collection of ShapeTrees[9] to constrain the *implicit* and *explicit* facts that are required by a dataset. A ShapeTree[10] in general, is used to express the layout of RDF Resources.

Another direction for future research is into the precise communication of reasoning rules / profiles that have been applied between systems; while introducing a minimal network overhead. This is necessary to make use-cases such as as those described in Section 9.3 robust. There has been some work in this area with the development of the Rule Interchange Format (RIF) (Kifer, 2008), however this work was performed prior to the development protocols like SOLID and needs to be adapted to meet the requirements of the associated ecosystem.

14.2 Next steps for Reasoning in T-FIRRE

Our work on T-FIRRE has focused on creating an architecture that guarantees that data (matching particular Quad Patterns) has the correct enrichment's applied to it *before* the data iterators are manipulated by the query operations. This excludes one common technique for enriching the results of a SPARQL query, known as query re-writing (Kontchakov and Zakharyaschev, 2014). In query re-writing, results are enriched by *modifying* the SPARQL query prior to execution, so that the results appear as though the original query were executed on the union of the implicit and explicit facts. For instance, if we have the rule `?s a ?o` ∧ `?o owl:subClassOf ?o2 -> ?s a`

[4] https://docs.inrupt.com/developer-tools/javascript/client-libraries/authentication/
[5] https://github.com/comunica/comunica-feature-solid/
[6] https://github.com/comunica/comunica-feature-link-traversal
[7] https://github.com/comunica/comunica/issues/958
[8] https://github.com/LDflex/LDflex/issues/15
[9] https://shapetrees.org/TR/specification/
[10] https://shapetrees.org/TR/primer/

`?o`, then the results of the query `SELECT * WHERE { ?s a ?o }` can implicitly include any implicit facts by re-writing the query to be `SELECT * WHERE { ?s a/owl:subClassOf* ?o }`. Whilst this work was beyond the time-scope of our project, it (at least on the surface) is a straight-forward addition in the architecture; simply requiring an additional bus that has the same interface to the Comunica query optimisation actor[11]. Actors implementing this query re-writing bus would then take a SPARQL Algebra[12] and Rules[13] as input and return an updated SPARQL Algebra that is re-written to include the requisite results.

Beyond these more immediate needs are many *exciting* opportunities to *extend* the work that is presented in this thesis and *integrate* it into orthogonal research. The most immediate opportunities lie in integrating the T-FIRRE engine with other developments/research occurring on the Comunica Engine - this includes points mentioned at the start of this thesis:

1. Link traversal research[14]
2. ShapeTrees research[15]
3. Data Interprobaility research[16]
4. The SPARQL 1.2 specification[17]
5. The development of sparqlee[18]
6. Integration with the Solid Ecosystem[19]

In order to better foster the interoperability of reasoners within the JavaScript/TypeScript ecosystem, there is also work required to standardise the *interfaces* of *rules* and *reasoners*. We have begun a discussion around this with the RDFJS community group[20] and hope to develop an official standard over time as the RDFJS Working Group begins[21].

We believe it is also important to continually improve the performance of all reasoning components, including those implemented within T-FIRRE and those that are implemented at lower levels of abstraction like in the N3.js index. Future work in this domain includes the performance improvements that are suggested in Section 12.6.

[11] https://github.com/comunica/comunica/tree/master/packages/bus-optimize-query-operation
[12] https://www.npmjs.com/package/sparqlalgebrajs
[13] https://github.com/comunica/comunica-feature-reasoning/blob/master/packages/reasoning-types/lib/rules.ts
[14] https://github.com/comunica/comunica-feature-link-traversal
[15] https://github.com/comunica/comunica-feature-link-traversal/tree/feature/shapetrees
[16] https://rml.io/
[17] https://comunica.dev/roadmap/
[18] https://github.com/comunica/sparqlee
[19] https://github.com/comunica/comunica-feature-solid
[20] https://github.com/rdfjs/types/issues/26
[21] https://github.com/rdfjs/community-group/issues/1

Finally, there are many features[22] which we have planned for the reasoning architecture itself. Some of the more developer ready additions to our architecture include:

1. Implement more actors for pushing down reasoning to lower abstraction layers, this includes using the C++ API for RDFox via the NodeJS C++ add-on[23]; to enable those that have a local RDFox database as one of their sources to make direct use of the reasoning components within the engine. We note that this particular example will only be of use to NodeJS applications and will not be able to be run in the browser.

2. Implementing native reasoning actors for more expressive rule profiles. Most pressing for a variety of use-cases,[24] is the development of an actor that can directly handle all N3Logic. The steps to achieve this are:

   a) Create a new rule interface for N3Rules and add it to the set of type definitions of Rules[25]. This interface will likely look similar to that proposed in the RDFJS standardisation discussion[26].

   b) Modify the existing N3 parser[27] to handle all N3 rules.

   c) Implement a reasoning actor on top of the reasoning bus[28], which specifically handles materialisation for N3 rules. A baseline version of this can be achieved by delegating the task of evaluating N3 rules back to the query engine using a construct query[29].

3. A reasoning delegation actor for generic RDFJS sources. Our work on N3.js has sparked a discussion around the addition of reasoning methods, or having a minimal Reasoning Wrapper to the RDFJS Store specification[30]. As this work progresses, we should also add an actor to the Comunica Engine that delegates the task of reasoning on *in-memory* RDFJS stores to those algorithms. Once the RDFJS reasoning work stabilises, the actual implementation of this actor is a matter of a few hours work due to the interoperable architecture we have built for reasoning in Comunica.

Some longer term developments are:

[22] https://github.com/comunica/comunica-feature-reasoning/issues
[23] https://nodejs.org/api/addons.html
[24] https://github.com/SolidLabResearch/Challenges/issues/35#issuecomment-1136735456
[25] https://github.com/comunica/comunica-feature-reasoning/blob/master/packages/reasoning-types/lib/rules.ts
[26] https://github.com/rdfjs/types/issues/26#issuecomment-980537568
[27] https://github.com/comunica/comunica-feature-reasoning/blob/master/packages/actor-rule-parse-n3/lib/ActorRuleParseN3.ts
[28] https://github.com/comunica/comunica-feature-reasoning/tree/master/packages/bus-rdf-reason
[29] https://github.com/comunica/comunica-feature-reasoning/blob/feat/faster-forwards-chaining/packages/actor-rule-evaluate-construct-query/lib/ActorRuleEvaluateConstructQuery.ts
[30] https://github.com/rdfjs/N3.js/pull/295#issuecomment-1125736669

1. Interoperable rule handling[31] - within the T-FIRRE architecture that we have developed, different reasoning actors may be capable of handling different reasoning profiles. With the actors that handle more restrictive reasoning profile, generally being more efficient due to the fact that they are able to make additional assumptions about the rules, due to the large number of rule profiles that are in existence. Over time, it becomes unwieldy to try and express all of the *different* profiles that a single actor can handle. Consequently, we need to take on a more intelligent approach, whereby we generate and maintain an ontology of reasoning profiles and the expressivity relationships that are between them. Using this information, we can then define only the most expressive profile that a particular actor implementation and apply inference over the rule hierarchy to determine if that actor can handle the rules in the reasoning task that is about to be performed. An orthogonal challenge to this is, handling rule-set subsumptions within a given reasoning profile.

2. Dynamically choose between incremental, lazy, eager reasoning - at present, the choice of reasoning technique is made based on the *configuration* of the reasoning engine that is used. This means that there is little *run-time* control over whether lazy, eager and/or incremental reasoning is applied. Future work is to add components that enable configurations where this can be chosen at run time. A more extended research challenge here, is for the reasoner to dynamically choose between these reasoning techniques. Further to this, there should also be long term development for dynamically selecting between re-writing vs. materialisation reasoning techniques and research into dynamically selecting between backward and forward chaining. The latter is the most challenging, as there is 'no principle to tell whether to use top-down or bottom-up reasoning'[32].

3. Producing / Validating proofs - In many use cases it is common that a proof is required to *explain* why given implicit facts were generated (Kravari et al., 2011). Reasons for this include enabling verification of results by other machines. Another reason may be that the *explicit causes* need to be verified by a human in order to ensure their *validity*. Either way, future work for the engine includes the addition of a *proof bus*[33], so as to enable reasoners to generate and validate proofs generated by reasoners.

4. Push down query information such as BGP patterns to limit the amount of reasoning that needs to take place. The current 'lazy reasoning' behaviour uses a single quad pattern to restrict the set of rules that are used in a given reasoning operation. This can be greatly restricted by using *more* information about the query, to further restrict the set of conclusions that one is interested in.

[31] https://github.com/comunica/comunica-feature-reasoning/issues/22
[32] https://josd.github.io/quote.html
[33] https://github.com/comunica/comunica-feature-reasoning/issues/52

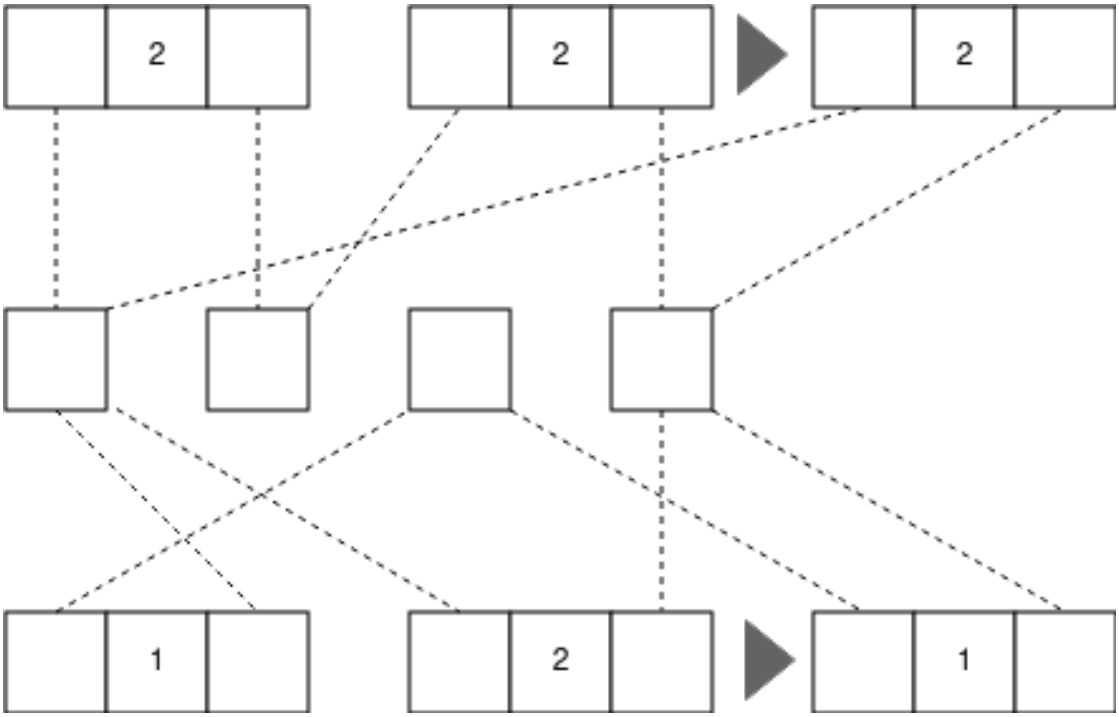

Figure 14.1: N3.js in-memory rules with shared memory variables

14.3 Next steps for Reasoning in N3.js

Despite achieving 'lightning-fast' browser side reasoning in N3.js; there is still much room to improve the performance of reasoning within the browser.

14.3.1 Shared memory rules

The first strategy that we propose for further improving the performance in N3.js is to share memory references between rules as shown in Figure 14.1. The consequence of this proposal is that the pre-filling step from Figure 11.6 no longer needs to be completed thus reducing the amount of computation that needs to be done in reasoning.

14.3.2 Reasoning on `.add`

Another useful feature for the N3.js store is in-built reasoning support, that is, support for reasoning as new data is added to the store. This is particularly important in use-cases when the store is being used to hold data loaded from remote sources; as reasoning can be applied while waiting for the remainder of the data to be retrieved from the remote source, rather than waiting for all data to arrive from the remote source before reasoning can be applied.

14.3.3 Long term goals

As web-assembly[34] (WASM) gains support, and receives performance optimisations it is worth transferring the optimisations here into a web-assembly module; which gives access to lower-level primitives and thus enables further performance optimisations to be made. In addition WASM is beginning to support *multi-threading* and GPU support may also be made available in browsers through WebGPU[35] - this will allow reasoning to be parallelised and thus see significant performance increases as a result.

14.4 Qualitative Evaluation

It was beyond the scope and resourcing of this project to perform a user-study of the T-FIRRE architecture that we have developed. It would be useful to do such a study in the future to refine the parts of the API that *extend* the current RDFJS API. This would also help inform design decisions made in future developments of RDFJS API as the Community Group explores standardised interfaces for the *invocation* of reasoners[36].

[34] https://webassembly.org/
[35] https://www.w3.org/TR/webgpu/
[36] https://github.com/rdfjs/types/issues/26

Appendix A

Appendix: Explanation on Appendices

A.1 Prefixes

Prefix	URI
ex	http://example.org/
foaf	http://xmlns.com/foaf/0.1/
owl	http://www.w3.org/2002/07/owl#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
timbl	https://timbl.inrupt.net/profile/card#

A.2 RDFS Rules

```
(?u ?a ?y) -> (?a a rdf:Property)

(?a rdfs:domain ?x) ^ (?u ?a ?y) -> (?u a ?x)

(?a rdfs:range ?x) ^ (?u ?a ?v) -> (?v a ?x)

(?u ?a ?x) -> (?u a rdfs:Resource)

(?u ?a ?v) -> (?v a rdfs:Resource)

(?u rdfs:subPropertyOf ?v) ^ (?v rdfs:subPropertyOf ?x)
    -> (?u rdfs:subPropertyOf ?x)

(?u a rdf:Property) -> (?u rdfs:subPropertyOf ?u)

(?a rdfs:subPropertyOf ?b) ^ (?u ?a ?y) -> (?u ?b ?y)

(?u a rdfs:Class) -> (?u rdfs:subClassOf rdfs:Resource)

(?u rdfs:subClassOf ?x) ^ (?v a ?u) -> (?v a ?x)

(?u a rdfs:Class) -> (?u rdfs:subClassOf ?u)

(?u rdfs:subClassOf ?v) ^ (?v rdfs:subClassOf ?x)
    -> (?u rdfs:subClassOf ?x)

(?u a rdfs:ContainerMembershipProperty)
    -> (?u rdfs:subPropertyOf rdfs:member)

(?u a rdfs:Datatype) -> (?u rdfs:subClassOf rdfs:Literal)
```

Listing A.1: The RDFS Rule Set

A.3 T-FIRRE interfaces

```
export interface IActionRdfReason {
  /**
   * The patterns for which must have all inferred data.
   *
   * If left undefined then all inferences on the data
   * need to be made.
   */
  pattern?: Algebra.Pattern;
  /**
   * Any quads that are currently being inserted/deleted.
   * This is used for incremental reasoning and stream
   * reasoning.
   */
  updates?: {
      quadStreamInsert?: AsyncIterator<RDF.Quad>;
      quadStreamDelete?: AsyncIterator<RDF.Quad>;
  };
  /**
   * Context data that should be forwarded to all actors.
   * This contains information including, the sources for
   * data that is being reasoned over, definitions
   * of reasoning groups, and reasoning rules.
   */
  context: IActionContext;
};
```

Figure A.1: Action interface for the Reasoning Bus

```
export interface IActionRdfResolveQuadPattern extends IAction {
  /**
  * The quad pattern to resolve.
  */
  pattern: Algebra.Pattern;
}

type IActionRdfResolveQuadPatternInterceptReasoned =
  IActionRdfResolveQuadPattern;

export interface IActorRdfResolveQuadPatternOutput extends
  IActorOutput {
  /**
  * The resulting quad data stream.
  *
  * The returned stream MUST expose the property 'metadata'.
  * The implementor is reponsible for handling cases where
  * 'metadata' is being called without the stream being in
  * flow-mode.
  *
  * This metadata object MUST implement IMetadata.
  * @see IMetadata
  */
  data: AsyncIterator<RDF.Quad>;
}

type IActorRdfResolveQuadPatternOutputInterceptReasoned =
  IActorRdfResolveQuadPatternOutput;
```

Figure A.2: Action and output interfaces for resolving quad patterns

```
export type IDatasetFactory = () => IDataSource & IDataDestination;

export interface IReasonedSource {
  type: 'full';
  reasoned: true;
  done: Promise<void>;
}

export interface IUnreasonedSource {
  type: 'full';
  reasoned: false;
}

export interface IPartialReasonedStatus {
  type: 'partial';
  patterns: Map<RDF.BaseQuad, IReasonStatus>;
}

export type IReasonStatus = IReasonedSource | IUnreasonedSource;

export interface IReasonGroup {
  dataset: IDataSource & IDataDestination;
  status: IReasonStatus | IPartialReasonedStatus;
  context: IActionContext;
}
```

Figure A.3: Interfaces to track the reasoning status

Bibliography

2018 reform of eu data protection rules. https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf. [Cited on page 3.]

Abiteboul, S.; Hull, R.; and Vianu, V., 1995. *Foundations of databases*, vol. 8. Addison-Wesley Reading. [Cited on page 72.]

Acosta, M.; Hartig, O.; and Sequeda, J., 2019. Federated rdf query processing. (2019). [Cited on page 22.]

Apt, K. R., 1990. Logic programming. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990 (1990), 493–574. [Cited on page 10.]

Arndt, D.; Meester, B. D.; Dimou, A.; Verborgh, R.; and Mannens, E., 2017. Using rule-based reasoning for rdf validation. In *Rules and Reasoning*, 22–36. Springer International Publishing, Cham. [Cited on page 49.]

Auer, S.; Bizer, C.; Kobilarov, G.; Lehmann, J.; Cyganiak, R.; and Ives, Z., 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*, 722–735. Springer. [Cited on page 21.]

Baader, F.; Calvanese, D.; McGuinness, D.; Patel-Schneider, P.; Nardi, D.; et al., 2003. *The description logic handbook: Theory, implementation and applications*. Cambridge university press. [Cited on page 16.]

Benbernou, S.; Huang, X.; and Ouziri, M., 2017. Semantic-based and entity-resolution fusion to enhance quality of big rdf data. *IEEE Transactions on Big Data*, 7, 2 (2017), 436–450. [Cited on page 27.]

Berners-Lee, T., 2002. Enabling standards and technologies - layer cake. https://www.w3.org/2002/Talks/04-sweb/slide12-0.html. Accessed: 2022–05-16. [Cited on pages xiii, 11, and 12.]

Berners-Lee, T.; Fielding, R.; and Masinter, L., 1998. Rfc2396: Uniform resource identifiers (uri): generic syntax. [Cited on page 74.]

Berners-Lee, T.; Hendler, J.; and Lassila, O., 2001a. The semantic web. *Scientific american*, 284, 5 (2001), 34–43. [Cited on pages 11 and 21.]

Berners-Lee, T.; Hendler, J.; and Lassila, O., 2001b. The Semantic Web. *Scientific American*, (2001). [Cited on page 3.]

Bewig, P. L., 2007. Scheme request for implementation 41: Streams. (2007). [Cited on page 88.]

Bizer, C.; Heath, T.; Idehen, K.; and Berners-Lee, T., 2008. Linked data on the web (ldow2008). In *Proceedings of the 17th international conference on World Wide Web*, 1265–1266. [Cited on page 3.]

Blanchette, J.-F., 2012. *Burdens of proof: Cryptographic culture and evidence law in the age of electronic documents*. MIT Press. [Cited on page 4.]

Brickley, D. and Miller, L., 2007. FOAF Vocabulary Specification 0.91. Namespace document. http://xmlns.com/foaf/spec/. [Cited on page 109.]

Broekstra, J. and Kampman, A., 2003. Inferencing and truth maintenance in rdf schema. *PSSS*, 89 (2003). [Cited on page 14.]

Brumm, B., 2019. Inner join. In *Beginning Oracle SQL for Oracle Database 18c*, 253–262. Springer. [Cited on pages 40 and 88.]

Capadisli, S.; Berners-Lee, T.; Verborgh, R.; and Kjernsmo, K., 2021. Solid protocol. *URL https://solidproject.org/TR/2021/protocol-20211217*, (2021). [Cited on pages 3 and 29.]

Carroll, J. J.; Bizer, C.; Hayes, P.; and Stickler, P., 2005. Named graphs. *Journal of Web Semantics*, 3, 4 (2005), 247–267. [Cited on pages 13 and 19.]

Cerami, M., 2014. An introduction to description logic v relations to modal logic. University Lecture. http://phoenix.inf.upol.cz/~ceramim/DL/DL05.pdf. [Cited on page 17.]

Ceri, S.; Gottlob, G.; Tanca, L.; et al., 1989. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1, 1 (1989), 146–166. [Cited on page 14.]

Chaston, I. et al., 2015. *Internet marketing and big data exploitation*. Springer. [Cited on page 3.]

Christophides, V.; Efthymiou, V.; and Stefanidis, K., 2015. Entity resolution in the web of data. *Synthesis Lectures on the Semantic Web*, 5, 3 (2015), 1–122. [Cited on page 27.]

Clocksin, W. F. and Mellish, C. S., 2003. *Programming in PROLOG*. Springer Science & Business Media. [Cited on page 10.]

Coburn, A.; elf Pavlik; and Zagidulin, D., 2022. Solid-oidc. *URL https://solidproject.org/TR/oidc*, (2022). [Cited on page 30.]

Consortium, U., 2015. Uniprot: a hub for protein information. *Nucleic acids research*, 43, D1 (2015), D204–D212. [Cited on page 3.]

Corcho, O. and Gómez-Pérez, A., 2000. A roadmap to ontology specification languages. In *International Conference on Knowledge Engineering and Knowledge Management*, 80–96. Springer. [Cited on page xx.]

Craig, B. L., 2007. The oo jdrew engine of rule responder: Naf hornlog ruleml query answering. In *Advances in Rule Interchange and Applications*, 149–154. Springer Berlin Heidelberg, Berlin, Heidelberg. [Cited on page 104.]

De Filippi, P. and McCarthy, S., 2012. Cloud computing: Centralization and data sovereignty. *European Journal of Law and Technology*, 3, 2 (2012). [Cited on page 3.]

De Giacomo, G. and Lenzerini, M., 1994. Concept language with number restrictions and fixpoints, and its relationship with mu-lculus. In *EI*, vol. 94, 411–415. Citeseer. [Cited on page 16.]

De Rijke, M. et al., 1998. Modal logics and description logics. *Proc. DL*, 98 (1998), 1–3. [Cited on page 17.]

Diallo, M. H., 2018. *User-Centric Security and Privacy Approaches in Untrusted Environments*. Ph.D. thesis, UC Irvine. [Cited on page 4.]

Elliott, E., 2020. *Composing software: An exploration of functional programming and object composition in JavaScript*. Lean Publishing. [Cited on page 82.]

Erling, O. and Mikhailov, I., 2009. Rdf support in the virtuoso dbms. In *Networked Knowledge-Networked Media*, 7–24. Springer. [Cited on page 21.]

Finin, T.; Fritzson, R.; Matuszek, D.; et al., 1989. Adding forward chaining and truth maintenance to prolog. *5th Artificial Intelligence Applications*, (1989), 123–130. [Cited on page 14.]

Gallier, J. H., 2015. *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications. [Cited on page 11.]

Gayo, J. E. L.; Prud'Hommeaux, E.; Boneva, I.; and Kontokostas, D., 2017. Validating rdf data. *Synthesis Lectures on Semantic Web: Theory and Technology*, 7, 1 (2017), 1–328. [Cited on page 68.]

Gholami, A. and Laure, E., 2016. Security and privacy of sensitive data in cloud computing: a survey of recent developments. *arXiv preprint arXiv:1601.01498*, (2016). [Cited on page 4.]

Glimm, B.; Horrocks, I.; Motik, B.; Stoilos, G.; and Wang, Z., 2014. HermiT: An OWL 2 Reasoner. *Journal of Automated Reasoning*, 53, 3 (Oct 2014), 245–269. https://doi.org/10.1007/s10817-014-9305-1. [Cited on pages 22 and 104.]

Gruber, T. R., 1993. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5, 2 (1993), 199–220. [Cited on page xx.]

Guo, Y.; Pan, Z.; and Heflin, J., 2005. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3, 2-3 (2005), 158–182. [Cited on page 108.]

Gupta, A.; Mumick, I. S.; and Subrahmanian, V. S., 1993. Maintaining views incrementally. *ACM SIGMOD Record*, 22, 2 (1993), 157–166. [Cited on page 15.]

Harald Karlsen, L., 2015a. Description logic 1: Syntax and semantics. University Lecture. https://www.uio.no/studier/emner/matnat/ifi/INF3170/h15/undervisningsmateriale/dl1.pdf. [Cited on page 17.]

Harald Karlsen, L., 2015b. Description logic 2: Reasoning. University Lecture. https://www.uio.no/studier/emner/matnat/ifi/INF3170/h15/undervisningsmateriale/dl2.pdf. [Cited on page 17.]

Hartig, O., 2013a. An overview on execution strategies for linked data queries. *Datenbank-Spektrum*, 13, 2 (2013), 89–99. [Cited on pages xx and 15.]

Hartig, O., 2013b. Squin: a traversal based query execution system for the web of linked data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 1081–1084. [Cited on page xx.]

Hewitt, C.; Bishop, P.; and Steiger, R., 1973. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, 235–245. [Cited on page 36.]

Hitzler, P.; Krötzsch, M.; Parsia, B.; Patel-Schneider, P. F.; Rudolph, S.; et al., 2009. Owl 2 web ontology language primer. *W3C recommendation*, 27, 1 (2009), 123. [Cited on page 109.]

Horn, A., 1951. On sentences which are true of direct unions of algebras1. *The Journal of Symbolic Logic*, 16, 1 (1951), 14–21. [Cited on page 10.]

Huang, S. S.; Green, T. J.; and Loo, B. T., 2011. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 1213–1216. [Cited on page 10.]

Isele, R.; Umbrich, J.; Bizer, C.; and Harth, A., 2010. Ldspider: An open-source crawling framework for the web of linked data. In *Proceedings of the 2010 International Conference on Posters & Demonstrations Track*, vol. 658, 29–32. Citeseer. [Cited on page xx.]

Keet, C. M., 2013. *Open World Assumption*, 1567–1567. Springer New York, New York, NY. ISBN 978-1-4419-9863-7. doi:10.1007/978-1-4419-9863-7_734. https://doi.org/10.1007/978-1-4419-9863-7_734. [Cited on page 17.]

Khamparia, A. and Pandey, B., 2017. Comprehensive analysis of semantic web reasoners and tools: a survey. *Education and Information Technologies*, 22, 6 (2017), 3121–3145. [Cited on page 21.]

Kifer, M., 2008. Rule interchange format: The framework. In *International Conference on Web Reasoning and Rule Systems*, 1–11. Springer. [Cited on page 118.]

Knublauch, H. and Kontokostas, D., 2017. Shapes constraint language (SHACL). W3C Recommendation, w3c. https://www.w3.org/TR/shacl. [Cited on pages 31, 44, 68, and 118.]

Kontchakov, R. and Zakharyaschev, M., 2014. An introduction to description logics and query rewriting. In *Reasoning Web International Summer School*, 195–244. Springer. [Cited on page 118.]

Kravari, K.; Papatheodorou, K.; Antoniou, G.; and Bassiliades, N., 2011. Reasoning and proofing services for semantic web agents. In *Twenty-Second International Joint Conference on Artificial Intelligence*. [Cited on page 121.]

Kurtz, D.; Parker, G.; Shotton, D.; Klyne, G.; Schroff, F.; Zisserman, A.; and Wilks, Y., 2009. Claros-bringing classical art to a global public. in escience, 20–27. *IEEE Computer Society*, (2009). [Cited on page 4.]

Le, H.; Boussetta, K.; and Achir, N., 2020. A unified and semantic data model for fog computing. In *2020 Global Information Infrastructure and Networking Symposium (GIIS)*, 1–6. IEEE. [Cited on page 5.]

Lloyd, J. W., 1994. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, 18–30. [Cited on page 10.]

Maarala, A. I.; Su, X.; and Riekki, J., 2016. Semantic reasoning for context-aware internet of things applications. *IEEE Internet of Things Journal*, 4, 2 (2016), 461–473. [Cited on page 15.]

Mallea, A.; Arenas, M.; Hogan, A.; and Polleres, A., 2011. On blank nodes. In *International semantic web conference*, 421–437. Springer. [Cited on page 41.]

Mansour, E.; Sambra, A. V.; Hawke, S.; Zereba, M.; Capadisli, S.; Ghanem, A.; Aboulnaga, A.; and Berners-Lee, T., 2016. A demonstration of the solid platform for social web applications. In *Proceedings of the 25th international conference companion on world wide web*, 223–226. [Cited on page 3.]

Mishra, R. B. and Kumar, S., 2011. Semantic web reasoners and languages. *Artificial Intelligence Review*, 35, 4 (2011), 339–368. [Cited on page 14.]

Motik, B.; Nenov, Y.; Piro, R.; and Horrocks, I. Handling owl: sameas in rdfox via rewriting. [Cited on pages 4, 74, and 75.]

Motik, B.; Nenov, Y.; Piro, R.; Horrocks, I.; and Olteanu, D., 2014. Parallel materialisation of datalog programs in centralised, main-memory rdf systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28. [Cited on pages 15 and 73.]

Motik, B.; Nenov, Y.; Piro, R. E. F.; and Horrocks, I., 2015. Incremental update of datalog materialisation: the backward/forward algorithm. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*. [Cited on pages 16 and 22.]

Motik, B.; Patel-Schneider, P. F.; Parsia, B.; Bock, C.; Fokoue, A.; Haase, P.; Hoekstra, R.; Horrocks, I.; Ruttenberg, A.; Sattler, U.; et al., 2009. Owl 2 web ontology language: Structural specification and functional-style syntax. *W3C recommendation*, 27, 65 (2009), 159. [Cited on page 13.]

Nah, F. F.-H., 2004. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23, 3 (2004), 153–163. [Cited on pages 23, 103, and 109.]

Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; and Banerjee, J., 2015. Rdfox: A highly-scalable rdf store. In *International Semantic Web Conference*, 3–20. Springer. [Cited on pages 10, 15, and 22.]

Olmedilla, D., 2007. Security and privacy on the semantic web. In *Security, privacy, and trust in modern data management*, 399–415. Springer. [Cited on page 4.]

Parravicini, A. and Mueller, R., 2021. The cost of speculation: Revisiting overheads in the v8 javascript engine. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 13–23. IEEE. [Cited on page 105.]

Pautasso, C. and Wilde, E., 2009. Why is the web loosely coupled? a multi-faceted metric for service design. In *Proceedings of the 18th international conference on World wide web*, 911–920. [Cited on page 3.]

Portoraro, F., 2001. Automated reasoning. (2001). [Cited on page 9.]

Prud'hommeaux, E.; Boneva, I.; Gayo, J. E. L.; and Kellogg, G., 2019. Shape expressions language 2.1. Final community group report 8 october 2019, W3C Community Group. http://shex.io/shex-semantics/. [Cited on pages 44, 68, and 118.]

Rattanasawad, T.; Saikaew, K. R.; Buranarach, M.; and Supnithi, T., 2013. A review and comparison of rule languages and rule-based inference engines for the semantic web. In *2013 International Computer Science and Engineering Conference (ICSEC)*, 1–6. doi:10.1109/ICSEC.2013.6694743. [Cited on page 14.]

Reyes-Alvarez, L.; Molina-Morales, D.; Hidalgo-Delgado, Y.; del Mar Roldán-Garcia, M.; and Aldana-Montes, J. F., 2014. Exploring incremental reasoning approaches based on module extraction. In *CEUR Workshop Proceedings*, vol. 1219, 1–12. Citeseer. [Cited on page 15.]

Robinson, J. A., 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12, 1 (1965), 23–41. [Cited on page 11.]

Russell, S. and Norvig, P., 2009. Artificial intelligence: A modern approach. edition. [Cited on page 15.]

Sakr, S.; Wylot, M.; Mutharaju, R.; Le Phuoc, D.; and Fundulaki, I., 2018. *Linked Data: Storing, Querying, and Reasoning*. Springer. [Cited on page 22.]

Sengupta, K. and Hitzler, P., 2014. Web ontology language (owl). *Encyclopedia of Social Network Analysis and Mining*, (2014). [Cited on page 17.]

Sima, A. C.; Mendes de Farias, T.; Zbinden, E.; Anisimova, M.; Gil, M.; Stockinger, H.; Stockinger, K.; Robinson-Rechavi, M.; and Dessimoz, C., 2019. Enabling semantic queries across federated bioinformatics databases. *Database*, 2019 (2019). [Cited on page 72.]

Sondes, T.; Elhadj, H. B.; and Chaari, L., 2019. An ontology-based healthcare monitoring system in the internet of things. In *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, 319–324. IEEE. [Cited on page 4.]

Stjernfelt, F. and Lauritzen, A. M., 2020. Facebook and google as offices of censorship. In *Your Post has been Removed*, 139–172. Springer. [Cited on page 3.]

Stuckenschmidt, H.; Ceri, S.; Della Valle, E.; and Van Harmelen, F., 2010. Towards expressive stream reasoning. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. [Cited on page 15.]

Suchanek, F.; Abiteboul, S.; and Senellart, P., 2011. Ontology alignment at the instance and schema level. *arXiv preprint arXiv:1105.5516*, (2011). [Cited on page 26.]

Taelman, R., 2022. *An overview of the most relevant buses and actors that are used in Comunica SPARQL*. https://comunica.dev/img/architecture_sparql.svg. [Cited on page 37.]

Taelman, R.; Van Herwegen, J.; Vander Sande, M.; and Verborgh, R., 2018. Comunica: a modular sparql query engine for the web. In *Proceedings of the 17th International Semantic Web Conference*. https://comunica.github.io/Article-ISWC2018-Resource/. [Cited on pages xiii, 6, 22, 37, and 38.]

Taelman, R.; Van Herwegen, J.; Vander Sande, M.; and Verborgh, R., 2022. Components.js: Semantic dependency injection. *Semantic Web Journal*, (Jan. 2022). https://linkedsoftwaredependencies.github.io/Article-System-Components/. [Cited on page 37.]

Terdjimi, M.; Médini, L.; and Mrissa, M., 2015. Hylar: Hybrid location-agnostic reasoning. In *ESWC Developers Workshop 2015*, 1. [Cited on pages 4, 5, 15, 22, 104, and 105.]

Terdjimi, M.; Médini, L.; and Mrissa, M., 2018. Web reasoning using fact tagging. In *Companion Proceedings of the The Web Conference 2018*, 1587–1594. [Cited on page 22.]

Tsarkov, D. and Horrocks, I., 2006. Fact++ description logic reasoner: System description. In *International joint conference on automated reasoning*, 292–297. Springer. [Cited on pages 18 and 22.]

Van Herwegen, J.; Taelman, R.; Capadisli, S.; and Verborgh, R., 2017. Describing configurations of software experiments as linked data. In *ISWC2017, the 16e International Semantic Web Conference*, vol. 1931, 1–8. [Cited on page 37.]

Verborgh, R. and De Roo, J., 2015. Drawing conclusions from linked data on the web: The eye reasoner. *IEEE Software*, 32, 3 (2015), 23–27. [Cited on pages 10, 22, and 104.]

Verborgh, R. and Taelman, R., 2020. Ldflex: A read/write linked data abstraction for front-end web developers. In *International Semantic Web Conference*, 193–211. Springer. [Cited on pages 3, 29, 47, 63, and 114.]

Verborgh, R.; Vander Sande, M.; Hartig, O.; Van Herwegen, J.; De Vocht, L.; De Meester, B.; Haesendonck, G.; and Colpaert, P., 2016. Triple pattern fragments: a low-cost knowledge graph interface for the web. *Journal of Web Semantics*, 37 (2016), 184–206. [Cited on pages 4, 22, 23, 39, 77, and 78.]

Vrandečić, D. and Krötzsch, M., 2014. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57, 10 (2014), 78–85. [Cited on pages 21 and 86.]

Werbrouck, J.; Pauwels, P.; Beetz, J.; and van Berlo, L., 2019. Towards a decentralised common data environment using linked building data and the solid ecosystem. In *36th CIB W78 2019 Conference*, 113–123. [Cited on page 4.]

Wood, D.; Lanthaler, M.; and Cyganiak, R. Rdf 1.1 concepts and abstract syntax, w3c recommendation, w3c (feb. 2014). *URL http://www. w3. org/TR/2014/REC-rdf11-concepts-20140225*. [Cited on pages 11, 22, and 41.]

Wright, J.; Méndez, S. J. R.; Haller, A.; Taylor, K.; and Omran, P. G., 2020a. on2ts-typescript generation from owl ontologies and shacl. In *ISWC (Demos/Industry)*, 358–363. [Cited on page 31.]

Wright, J.; Méndez, S. J. R.; Haller, A.; Taylor, K.; and Omran, P. G., 2020b. Schimatos: a shacl-based web-form generator for knowledge graph editing. In *International Semantic Web Conference*, 65–80. Springer. [Cited on page 31.]

Yi, S.; Hao, Z.; Qin, Z.; and Li, Q., 2015. Fog computing: Platform and applications. In *2015 Third IEEE workshop on hot topics in web systems and technologies (HotWeb)*, 73–78. IEEE. [Cited on page 4.]