School of Systems Engineering

Investigating Elastic Cloud Based RDF Processing

by

Omer Dawelbeit

Thesis submitted for the degree of Doctor of Philosophy

School of Systems Engineering

April 2016

University of Reading

Abstract

The Semantic Web was proposed as an extension of the traditional Web to give Web data context and meaning by using the Resource Description Framework (RDF) data model. The recent growth in the adoption of RDF in addition to the massive growth of RDF data, have led numerous efforts to focus on the challenges of processing this data. To this extent, many approaches have focused on vertical scalability by utilising powerful hardware, or horizontal scalability utilising always-on physical computer clusters or peer to peer networks. However, these approaches utilise fixed and high specification computer clusters that require considerable upfront and ongoing investments to deal with the data growth. In recent years cloud computing has seen wide adoption due to its unique elasticity and utility billing features.

This thesis addresses some of the issues related to the processing of large RDF datasets by utilising cloud computing. Initially, the thesis reviews the background literature of related distributed RDF processing work and issues, in particular distributed rule-based reasoning and dictionary encoding, followed by a review of the cloud computing paradigm and related literature. Then, in order to fully utilise features that are specific to cloud computing such as elasticity, the thesis designs and fully implements a **Cloud**-based Task **Ex**ecution framework (CloudEx), a generic framework for efficiently distributing and executing tasks on cloud environments. Subsequently, some of the large-scale RDF processing issues are addressed by using the CloudEx frame-

work to develop algorithms for processing RDF using cloud computing. These algorithms perform efficient dictionary encoding and forward reasoning using cloud-based columnar databases. The algorithms are collectively implemented as an **E**lastic **C**ost **A**ware **R**easoning **F**ramework (ECARF), a cloud-based RDF triple store. This thesis presents original results and findings that advance the state of the art of performing distributed cloud-based RDF processing and forward reasoning.

Declaration

I confirm that this is my own work and the use of all material from other sources has been properly and fully acknowledged.

Omer Dawelbeit

Dedication

To Amna and Ibrahim, my wonderful mother and father for their everlasting love, support and motivation.

Acknowledgements

"In the name of God, the Most Gracious, the Most Merciful"

All praise be to my God and Lord who created me and taught me that which I knew not, who helped me get through difficult times.

Being part-time, the PhD journey has been a long one for me, but it was the journey of a lifetime, a journey of discovery and exploration. Since I was young, doing a PhD was my dream, to be like my dad whose PhD journey has inspired the whole family. There are many people that supported, motivated and inspired me throughout this journey.

Firstly, I would like to thank my supervisor, Professor Rachel McCrindle, her kind encouragements and motivation gave me the strength and energy needed to complete the journey, her guidance made the path very clear.

I'm ever so grateful to my parents Amna and Ibrahim for their supplications and for teaching me to always seek knowledge and aspire to achieve. I would like to thank all my brothers and sisters for looking up to me as their elder brother, Emad, Mariam, Mohammed, Huda, Tasneem, Yousra and Osman. I would like to thank Osman in particular for the interesting discussions and for patiently listening to me talk about my research challenges.

My special thanks go to my companions in this journey, my wife Amna and my children Laila, Sarah and Muhammad. Amna has been the fuel of this journey with her love, patience, support and understanding, despite the fact that she was also busy on her own PhD. My children have inspired me to carry on and achieve to make them proud. They have been patient and lived with me through every moment of this journey, even Muhammad, my 4 years old son regularly used to ask me how the thesis chapters are progressing.

I would like to thank Google for their great technologies, developer support and documentations. Google support kindly and promptly lifted any quota restrictions that I requested to complete the evaluation of this research. They have kindly provided free credits to use the Google Cloud Platform.

Research Outcome

Publications

- *'ECARF: Distributed RDFS Reasoning in the Google Cloud'* - IEEE Transactions on Cloud Computing Journal - Jun 2015. Submitted.

Conference Papers

- *'A Novel Cloud Based Elastic Framework for Big Data Preprocessing'* - Sixth Computer Science and Electronic Engineering Conference 2014. Proceedings to be published by the IEEE - Sep 2014.
- *'Efficient Dictionary Compression for Processing RDF Big Data Using Google BigQuery'* - IEEE Communications Conference (Globecom 2016) - Dec 2016. Submitted.

Posters

- *Investigating Elastic Cloud Based Reasoning for the Semantic Web* - University of Reading postgraduate conference poster - Jul 2014.

Presentations

- The British Computer Society doctoral consortium presentation - May 2014.
- University of Reading 3 minutes thesis competition presentation - Jul 2014.

- PhD proposal accepted for presentation at the International Semantic Web Conference (ISWC) 2014 - Oct 2014.
- *'Google BigQuery, processing big data without the infrastructure'* - Reading & Thames Valley GDG group - Aug 2015.

Recognitions and Awards

- Recognised by Google as a Google Developer Expert on the Google Cloud Platform - Nov 2015.
- Supplied with monthly Google Cloud Platform credit.

Open Source Contributions and Impact

- ECARF[1], Elastic cloud-based RDF triple store for RDF reasoning and RDFS reasoning.
- CloudEx[2], A generic elastic cloud based framework for the execution of embarrassingly parallel tasks.

[1] `http://ecarf.io`
[2] `http://cloudex.io`

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The World Wide Web is a collection of interlinked hypertext documents designed primarily for humans to read, with search engines being used to crawl and index these pages to provide search capability. However, this search is usually based on keyword matching, rather than the meaning of the content. For example, searching to get an intelligent answer to the question *"Does John work at iNetria?"* would simply return Web pages with text that matches the words in the question. There is therefore, a need for a mechanism to formally express the meaning of data on the Web, hence the Semantic Web was proposed [2]. In 2001, Berners-Lee et al. outlined the vision of the Semantic Web as follows:

> "The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation"

1.1 The Semantic Web

The Semantic Web was proposed as an extension of the traditional Web to give Web data context and meaning (semantics). Consequently, intelligent software applications

that can parse these semantics can be developed to assist humans with many tasks. One of these tasks is question answering, to answer questions like the one posed previously. A few other examples include semantic search, social discovery, content enrichment and publishing, etc., to mention just a few. For the Semantic Web vision to become a reality, the meaning of data must be expressed in a formal way. This requires the definition of common vocabulary and formal semantics that both humans and software applications can understand. Software applications can then understand the meaning of this data and deduce new knowledge from it. Consider for example the following statements available on the Web and represented in plain English:

- John is an employee
- John works at iNetria
- John's full name is "John Smith"

It is easy for humans to understand these statements and to deduce that the individual named John is a person working at a business named iNetria. A software application, however, will not be able to deduce such implicit knowledge straightaway. These statements need to be represented in a formal language that the application understands. Additionally, the application needs to be supplied with a set of rules on how to deduce that extra implicit knowledge. Another issue that might face such an application is the context as to which John the user have in mind, there might be many. To express such knowledge, the Resource Description Framework (RDF) [3] data model can be used.

1.2 The Resource Description Framework

RDF provides a data model that can be used on the Web to add meaning to existing data such that it can be understood by both software applications and humans. The

RDF Schema (RDFS) [4], is a semantic extension of RDF that enables users to define a group of related resources and the relationships between them. Information in RDF is primarily represented using XML [5], however other compact and readable representations also exist such as the Notation3 (N3) [6] syntax. Resources in RDF can be identified by using Uniform Resource Identifiers (URIs), for example the individual named John can be identified by using `<http://inetria.org/directory/employee/smithj>`. To avoid using long URIs these are usually shortened using namespaces such as `employee:smithj`.

To demonstrate these concepts, consider for example the plain English statements presented previously, it can be said that the individual John is of type *Employee* and the entity iNetria is of type *Company*. It can also be said that the relationship between an *Employee* and a *Company* is represented through the *works at* relationship. These statements of knowledge — usually known as triples — contain three parts, a *subject*, *predicate* and an *object* and can be written as follows:

```
- employee:smithj rdf:type inetria:Employee
- business:inetria rdf:type inetria:Company
- employee:smithj inetria:works_at business:inetria
```

Usually, basic explicit knowledge about resources is encoded as RDF by humans then software applications are used to deduce further implicit knowledge by using rule-based reasoning. Rule-based reasoning requires that a set of rules — such as RDFS entailment rules [7] — to be applied repeatedly to a set of statements to infer new knowledge. For example, given the extra information that an *Employee* is a sub class of *Person* an application can use rule-based reasoning to deduce that the employee `employee:smithj` is actually of type *Person*. This knowledge may seem trivial to humans, however for software applications inferring such knowledge requires compu-

tational resources, with more statements and rules this task can become computationally difficult.

1.2.1 RDF Data on the Web

There is currently a great deal of community, organisations and governments[1] effort to provide RDF datasets on the Web covering a wide range of topics, such as government data, publications, life sciences, geographic, social web to mention just a few. Additionally, ongoing effort is also focusing on connecting pieces of information on the Semantic Web data, widely known as Linked Data[2]. As of 2014[3] this Linked Data cloud spanned more than 1014 publicly available datasets. To give an indication on the size of these datasets, which is usually measured in terms of the number of statements (RDF triples) in each dataset, consider the DBpedia[4] [8] dataset, which is a collection structured data extracted from Wikipedia. This dataset contains more than 3 billion statements covering 125 languages. Another example is the Linked Life Data[5] (LLD) that has 10 billion statements, providing a collection of biomedical data such as genes, protein, disease, etc. . . .

1.2.2 Use Cases

Many applications use Semantic Web technologies in particular RDF to provide capabilities such as semantic search, content aggregation and discovery. Perhaps some of the notable usages of these technologies for content enrichment and publishing were the BBC websites for both the 2010 World Cup [9] and 2012 Olympics [10]. Using these technologies the BBC was able to automatically aggregate web pages that con-

[1] `https://data.gov.uk/`
[2] `http://linkeddata.org/`
[3] `http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/`
[4] `http://wiki.dbpedia.org/Datasets`
[5] `http://linkedlifedata.com/`

tain links to relevant resources with minimal human management. Such applications usually utilise a special type of database for storing and managing RDF triples called *triple stores*, which provide the ability to load, update and delete RDF statements. Additionally, triple stores can perform forward rule-based reasoning to infer additional knowledge and support the query of the stored statements using the Protocol and RDF Query Language (SPARQL) [11]. For applications to be able to process, integrate or query large datasets such as DBpedia and Linked Life Data, high specification hardware and expensive software are usually required. To make the matter worse, these datasets are constantly growing, forcing users to constantly upgrade their software and hardware in order to keep up with the data growth.

1.3 The Growth to Big Data

Initially, RDF data was processed using centralised systems that run on a single computer, however, due to the constant growth of the Semantic Web RDF data, this data can now be described as Big Data [12]. To this extent, the Semantic Web Challenge [13] which is concerned with building end-user Semantic Web applications, was formerly known as Billion Triples Challenge is now renamed to Big Data Challenge. This data growth means that the average size of RDF datasets that an application needs to process well exceeds a billion statements of knowledge. Therefore, the computational processing power required to process these datasets — such as deducing implicit knowledge through rule-based reasoning — far exceeds the capabilities of a single computer. Consequently, commercially available large triple stores [14] have focused on solutions that utilise high specifications hardware.

Moreover, recent research work on large scale RDF processing in general and RDFS reasoning in particular, has moved from focusing on centralised algorithms to focusing

on distributed and parallel algorithms. These algorithms utilise either peer-to-peer networks using Distributed Hash Tables (DHT) [15, 16], or computer clusters using MapReduce [17] and other programming models [18, 19, 20, 21]. Aforementioned work has primarily focused on addressing the challenges of efficient data partitioning and assimilation between the various computing resources, which in most cases, are either part of an always-on computer cluster, or peer to peer network.

1.4 Challenges with Physical Computing

Some of the existing algorithms on large scale RDF processing utilise high specifications and dedicated computer clusters [19, 20, 21] that require large upfront infrastructure investments. These clusters usually contain a fixed number of always-on computing resources that are billed for 24/7 regardless of utilisation. This presents a number of challenges, for example, it is not possible to simply "switch off" these resources when not needed and then switch them on when there is enough workload. Besides, jobs executed on such clusters are constraint to the maximum number of computing resources available, which means larger jobs utilising all available resources will need to execute over longer time periods. Increasing the capacity of such clusters requires further investment and effort to purchase, install and configure additional computing resources.

Other algorithms that perform distributed RDF processing utilise computers in peer to peer networks [15, 16], namely Distributed Hash Tables [22] (DHTs), in these settings computing resources perform multiple tasks that are not related. In such heterogeneous computing environments, there are no performance or deadline guarantees to any one particular task. Moreover, due to the shared nature of such environments, it is difficult to measure the overall cost exclusively required to perform one particular

task. In addition to the aforementioned challenges, the Semantic Web data is often highly skewed which causes a number of workload partitioning strategies to suffer from load balancing issues [19]. Furthermore, the continuous growth of the Semantic Web data presents the challenges of storing both the explicit and the inferred data. Some of the issues described in this section are inherit from the nature of physical computing, cloud computing on the other hand is a promising computing paradigm that can be utilised to address some of these issues.

1.5 The Potential of Cloud Computing

Traditional distributed or grid computing is based on fixed, physical and interconnected servers hosted in data centres, cloud computing on the other hand is a computing paradigm built on top of these servers. Public cloud computing provides a pool of shared resources that can be rapidly acquired or released with minimal effort, either manually or programmatically [23], a feature unique to cloud computing termed Elasticity [24, 25, 26]. Elasticity is the ability to acquire computing resource with varying configurations (e.g. number of CPU cores, RAM size, disk type) almost instantly when needed and released instantly when not needed by using an Application Programming Interface (API) exposed by the cloud provider. Public cloud providers, most notably Amazon Web Services [27], Google Cloud Platform [28] and Microsoft Azure [29], utilise their data centres to commercially offer on-demand computing resources.

In addition to elasticity, the other key benefit of cloud computing is usage-based billing so that consumers only pay for what they use, which is highly cost efficient compared to a physical computer environment being billed even when the resources are not being used. Cloud computing offers many other on demand services, such as low latency

mass cloud storage capable of storing unlimited amounts of data and analytical big data services. These big data services such as Amazon Redshift [30] and Google BigQuery [31] provide a promising platform that is able to readily analyse web-scale datasets in a few seconds. For example, it was shown that the algorithms behind Google BigQuery are able to achieve a scan throughput of 100 billion records per second on a shared cluster, which out performs MapReduce by an order of magnitude [32]. Additionally, it was shown that using public clouds, users can scale to over 475 of computing nodes in less than 15 minutes [33].

1.6 Motivation

As noted in the previous section, the cloud computing paradigm provides a promising approach for storing, managing and processing large RDF datasets. Furthermore, cloud computing is both available and affordable to users and organisations, whereas, dedicated physical computing clusters may not be practically available for everyone to use due to the required upfront infrastructure investment and ongoing maintenance cost. Currently end-user applications that process, use and query large datasets such as DBpedia and Linked Life Data require high specification hardware to run commercially available triple stores, requiring massive upfront infrastructure investments. Utilising an approach based on cloud computing can play a step forward towards making RDF processing available for mainstream use, without any expensive upfront investment. Despite this, little research has been done to explore the use of cloud computing for RDF processing, storage and performing RDFS rule-based reasoning. It may be tempting to replicate approaches developed in physical computing on cloud computing, such approaches are primarily centred on computing clusters with the storage and processing contained therein. In doing so, it is very likely that such

approaches will suffer from reduced performance due to the virtualisation nature of cloud resources [20], which is yet another layer on top of the native computing resources. Instead of such replication, this thesis utilises and introduces the notion "*cloud-first frameworks*" to signify frameworks that are entirely cloud based, specifically designed from the ground-up for cloud computing and utilise cloud services such as Infrastructure as a Service, cloud storage and Big Data services.

1.7 Research Aim and Questions

Based on the challenges and motivation noted in the previous sections, the aim of this research is to develop and evaluate an elastic cloud-based triple store for RDF processing in general and RDFS rule-based reasoning in particular. This aim has resulted in a number research questions being asked, firstly, if the processing of RDF is performed on cloud computing resources, can these resources be efficiently acquired for task execution and released. Secondly, as noted earlier that RDF contains long URIs which occupy storage space, network bandwidth and take longer to process. A question worth asking is, can these URIs be compressed rapidly and efficiently by using an efficient dictionary encoding. Finally, provided that cloud big data service such as Google BigQuery provide analytical capabilities for massive datasets, can these services be used to perform rule-based reasoning. These research questions can be summarised as follows:

- Q1. How can a cloud-based system efficiently distribute and process tasks with different computational requirements?
- Q2. How can an efficient dictionary that fits in-memory be created for encoding RDF URIs from strings to integers?

- Q3. How can cloud-based big data services such as Google BigQuery be used to perform RDFS rule-based reasoning?

An overarching research question that encompasses all the above question is: can cloud computing provide an efficient RDF triple store that provides storage, processing and can perform RDFS rule-based reasoning?

1.7.1 Scope of Research

To provide answers to the research questions, this research is conducted over two domains, RDF and cloud computing as shown in Figure 1.1. The research develops a **Cloud**-based Task **Ex**ecution framework (CloudEx), which is a generic cloud-first framework for efficiently executing tasks on cloud environments. Subsequently, the CloudEx framework is used to develop algorithms for providing an RDF triple store on the cloud, these algorithms are collectively implemented as an **E**lastic **C**ost **A**ware **R**easoning **F**ramework (ECARF). Both CloudEx and ECARF provide answers to the research question mentioned in the previous section.

1.7.2 Impact of Research

This thesis presents original results and findings that advance the state of the art of distributed cloud-based RDF processing and forward RDFS reasoning. The CloudEx framework developed as part of this thesis enables users to run generic tasks with different computational requirements on the cloud. Additionally, the ECARF triple store described in this thesis makes it possible to commercially offer a pay-as-you-go cloud hosted RDF triple store, by using a Triple store as a Service (TaaS) model similar to Software as a Service. Furthermore, these approaches enable applications to harness the powers of the Semantic Web and RDF without any upfront investment

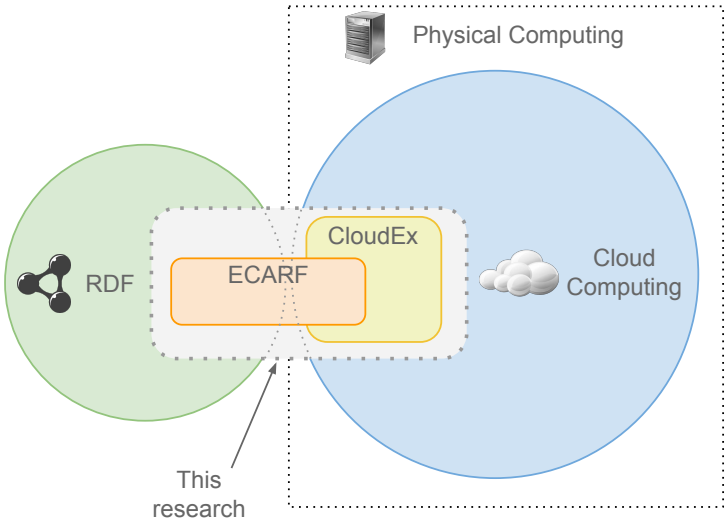


Figure 1.1: Research Context.

in expensive hardware or software. Some of the findings of this research have already attracted a great deal of interest (more than 10,000 views over a few days) from the technical community, in particular to potential savings with public clouds that charge per-minutes versus the ones that charge per-hour[6].

1.8 Technical Approach

To accomplish the aforementioned research aim and answer the research questions, this research follows the general methodology of design science research as outlined by [1] and summarised in Figure 1.2. As shown in the figure, the research is conducted over five iterative phases and can be summarised as follows:

[6] http://omerio.com/2016/03/16/saving-hundreds-of-hours-with-google-compute-engine-per-minute-billing/

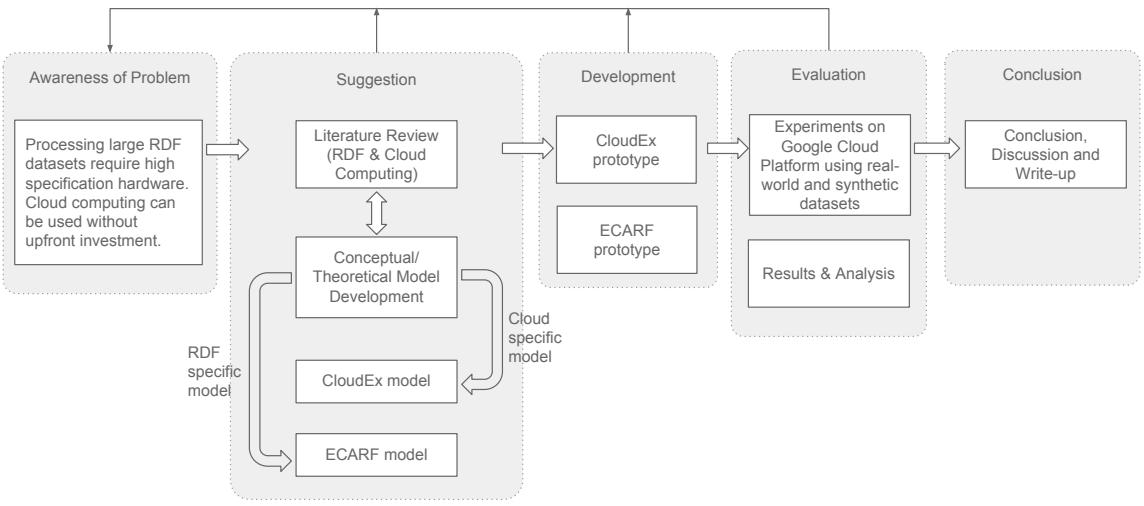


Figure 1.2: Research Methodology and Phases.

1. Literature review:

   - **RDF literature** - to review and identify the issues with approaches developed for distributed RDF processing and RDFS reasoning both in peer-to-peer networks and computer clusters.
   - **Cloud computing literature** - to review key services provided by public cloud providers with a review of the Google Cloud Platform [28], additionally, to survey and categorise related work on utilising cloud computing.

2. Conceptual / theoretical model development:

   - **CloudEx model** - To design a generic cloud-based architecture for tasks execution and workload partitioning on public clouds.
   - **ECARF model** - To design algorithms based on CloudEx for cloud-based RDF processing and performing RDFS forward reasoning using columnar databases.

3. Prototype development: to develop both the CloudEx and ECARF prototypes.

4. Evaluation:
    - To run experiments on the Google Cloud Platform using both real-world and synthetic datasets.
    - To gather an analyse the results.
5. Conclusion: to discuss the finding, draw conclusions and suggest future work.

1.9 Contributions

The contributions of this thesis are summarised as follows:

- This thesis has designed, implemented and evaluated CloudEx, a generic cloud based task execution framework that can be implemented on any public cloud. It is shown that CloudEx can acquire computing resources with various configurations to efficiently execute RDF tasks on the cloud, then release these resource once the work is done. CloudEx has successfully utilised many cloud services and acquired 1,086 computing resources with various specifications at a total cost of $290.29. The CloudEx framework is implemented as a publicly available open source framework[7].
- This thesis has designed, implemented and evaluated a dictionary encoding algorithm to generate an efficient dictionary for compressing RDF Uniform Resource Identifiers (URIs) using cloud computing resources. It is shown that this approach generates an efficient dictionary that can readily fit in-memory during the encoding and decoding process, which speeds up these processes considerably in comparison with other work. Additionally, the dictionaries generated

[7] `https://cloudex.io/`

using this approach are smaller than the ones created by other approaches that utilise dedicated hardware.

- This thesis has presented an algorithm to perform RDF processing and RDFS rule-based reasoning using distributed computing resources and big data services, in particular Google BigQuery [31]. It is shown that this approach is promising and provides the benefit of permanent storage for RDF datasets with the potential to query and develop a full triple store capability on top of BigQuery. In comparison, other related work output RDF datasets to files, with the data yet to be imported into a permanent storage with triple store capability.

- The last two contributions are implemented as ECARF, a cloud based RDF triple store with both processing and forward RDFS reasoning capabilities. ECARF is implemented on the Google Cloud Platform as a publicly available open source framework[8]. ECARF is the first to run RDFS reasoning entirely on cloud computing, unlike other approaches that simply replicate physical computing setup — such as MapReduce [20] — on the cloud, ECARF is entirely based on cloud computing features and services.

1.10 Outline of the Thesis

The outline of this thesis is shown in Figure 1.3. The remainder of the thesis is organised as follows, Chapters 2 and 3 provide background literature review on the Resource Description Framework and cloud computing respectively. Chapter 2 introduces the key concepts of RDF, RDFS and other key definitions that are used throughout this thesis. Additionally, the chapter provides a survey of related distributed RDF processing work and issues, in particular around distributed RDFS reasoning. Chapter 3

[8] `http://ecarf.io`

provides a review of the cloud computing paradigm including a high level overview of public cloud deployments. The key cloud services utilised in this research are reviewed with an introduction to the key Google Cloud Platform services. A background on some of the related research and commercial work utilising cloud elasticity and utility billing features is also provided. Chapter 4 introduces the design methodology used in the model development for both the CloudEx framework and the ECARF triple store.

Chapters 5, 6 and 7 present the original contributions and results of this thesis. Chapter 5 fully introduces the architecture of the CloudEx framework and present its various components. Subsequently, Chapter 6 presents a number of algorithms for RDF processing and rule-based RDFS reasoning using the CloudEx framework. These algorithms collectively define the ECARF RDF cloud-based triple store. Then, Chapter 7 presents the results and discussions of the experimental evaluation of both CloudEx and ECARF. Finally, Chapter 8 summarises the achievements of this thesis, discusses open issues and presents possible future work.

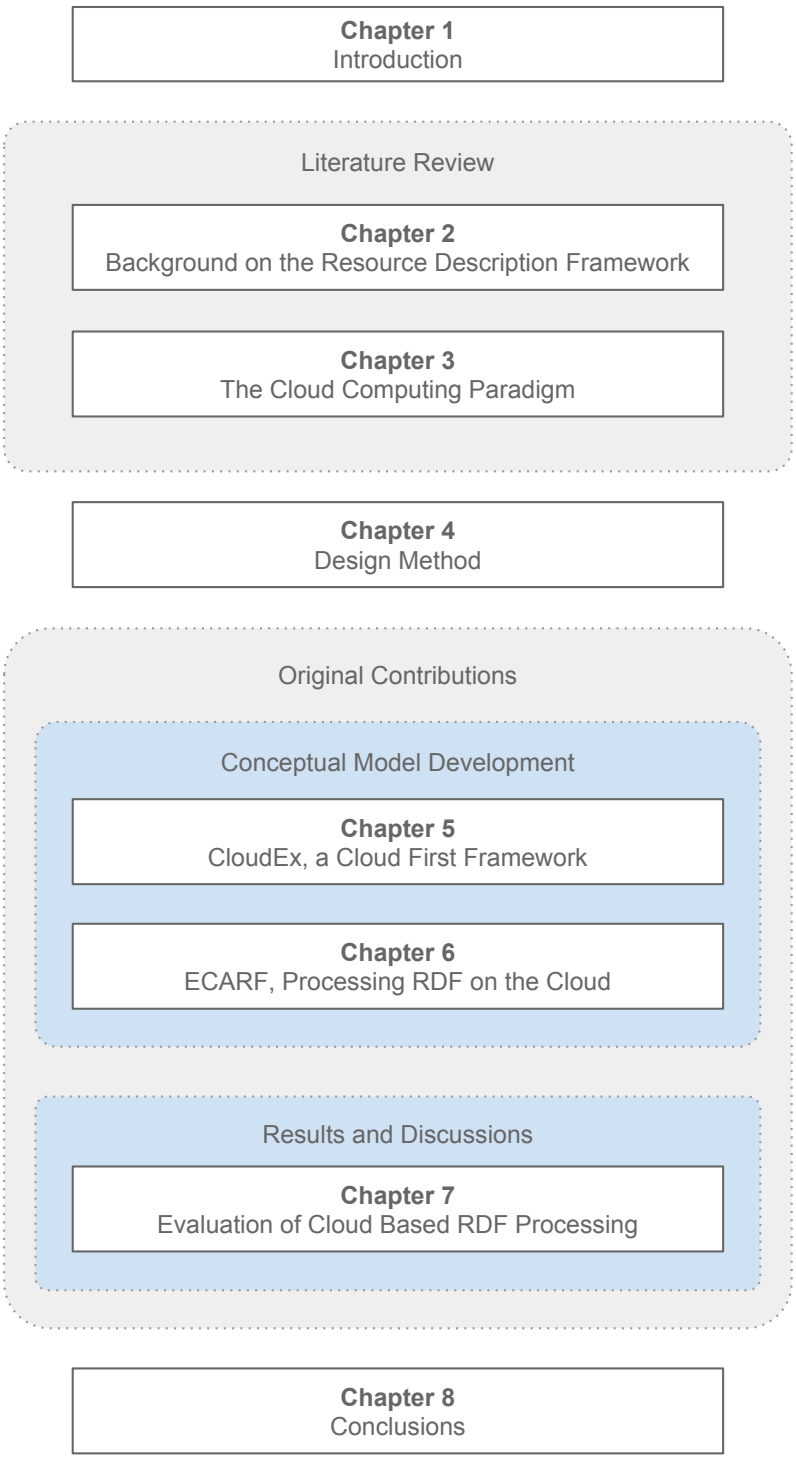


Figure 1.3: Thesis Outline.

Chapter 2

Background on the Resource Description Framework

The Resource Description Framework in addition to cloud computing are central to the work done in this research as was illustrated in Figure 1.1. More specifically, the primary aim of this research is to develop and evaluate an RDF triple store for RDF processing in general and RDFS rule-based reasoning in particular using cloud computing. The focus on RDF is motivated by the recent growth in the adoption of Semantic Web and RDF technologies, which is evident from efforts like Linked Data[1] and Schema.org[2]. Linked Data focuses on connecting RDF data on the Web and Schema.org focuses on providing common schemas for structured data on the Web using RDF and other frameworks. Schema.org for example, is sponsored by major search engines providers like Google, Microsoft and Yahoo to enable search engines to intelligently understand the meaning of structured data on the Web.

Another motivation for this research is the explosive growth of RDF datasets on the Web, which has also led to numerous other research and commercial efforts to effec-

[1] `http://linkeddata.org/`
[2] `http://schema.org/`

tively process this data. With large RDF datasets of a billion and more statements, efforts have focused on using distributed computing on dedicated computer clusters, however this is not without challenges. For example, the challenges of dealing with the ever increasing storage and processing requirements for such big data. Additionally, most research effort on large scale distributed RDF reasoning does not provide a mechanism to be able to query this data, which is a key requirement for applications to make use and benefit from such data.

This chapter is the first of the literature review chapters and introduces the key concepts of RDF that are used in this research to develop ECARF, furthermore, a survey of related literature is also provided. Firstly, in Section 2.1, some of the RDF key concepts and definitions used throughout this thesis are introduced, such as the RDF Schema (RDFS) and RDFS reasoning. Additionally, the concept of *ontology* is also introduced. Then in Section 2.2, a review of the literature covering the distributed processing and management of RDF and RDFS is provided.

Section 2.3 reviews RDF data management efforts that utilise cloud computing, subsequently in Section 2.4 a review of large scale RDF dictionary encoding literature is provided. Followed by Section 2.5 that summarises the challenges facing large scale RDF processing presented in this chapter. Finally, in Section 2.5, this chapter is concluded with a summary of the issues and challenges with the approaches surveyed herein.

2.1 Resource Description Framework

RDF [3] is recommended by the W3C for representing information about resources in the World Wide Web. RDF is intended for use cases when information need to be processed by and exchanged between applications rather than people. Information in

RDF is primarily represented in XML, however other compact representations also exist such as the Notation3 (N3) [6]. Resources in RDF can be identified by using Uniform Resource Identifier (URI) references (URIrefs), for example:

`<http://inetria.org/directory/employee/smithj>`

As a convention, to avoid writing long URIrefs, they are shortened using namespaces to replace the common part of the URI with a prefix. The previous example can be written as `employee:smithj` by using the prefix `employee:` to replace the URI `<http://inetria.org/directory/employee/>`.

Unknown resources or resources that do not need to be explicitly identified are called blank nodes. Blank nodes are referenced using an identifier prefixed with an underscore such as `_:nodeId`. Constant values such as strings, dates or numbers are referred to as literals. Information about a particular resource is represented in a statement, called a triple, that has the format $(subject, predicate, object)$ abbreviated as (s, p, o). Subject represents a resource, either a URI or a blank node, predicate represents a property linking a resource to an object, which could be another resource or literal. More formally let there be pairwise disjoint infinite sets of URIrefs (U), blanks nodes (B), and literals (L). An RDF triple is a tuple:

$$(s, p, o) \in (U \cup B) \times (U) \times (U \cup B \cup L) \tag{2.1}$$

A set of RDF triples is called an RDF graph (Figure 2.1 (a)), in which each triple is represented as a node-arc-node link. The node (ellipse or rectangle) represents the subject and object, the arc represents the predicate and is directed towards the object node. RDF triples can be exchanged in a number of formats, primarily RDF/XML [5] which is based on XML documents. Other formats include line based, plain text encoding of RDF graphs such as N-Quads [34] and N-Triples [35], which are simplified

subsets of N3. RDF defines a built-in vocabulary of URIrefs that have a special meaning. For example, to indicate that a resource is an instance of a particular kind or class, RDF defines the predicate `http://www.w3.org/1999/02/22-rdf-syntax-ns\#type`, shortened as `rdf:type`.

Figure 2.1 shows a running example of a simple RDF dataset of employees and their relationships. The RDF graph (Figure 2.1 (a)) shows URIrefs resources as ellipses with text, blanks nodes as empty ellipses and literals as rectangles. The example uses a number of well-known vocabularies such as RDF (`rdf:`), RDFS (`rdfs:`), which will be explained in the next section and Friend of a Friend[3] (`foaf:`). The example also defines a custom vocabulary (`inetria:`), which is summarised in Figure 2.1 (c).

The RDF graph describes a number of resources, mainly an employee (`employee:smithj`) and his properties, such as name (`foaf:name`) and start date (`inetria:start_date`), which are both literals. The example also describes properties that refer to other resources rather than literals, such as *manages* (`inetria:manages`) and *works at* (`inetria:works_at`). The RDF triples in Figure 2.1 (c) and (d) are represented in N-Triples format, with the long URI namespaces shortened with the prefixes shown in Figure 2.1 (b) for readability.

2.1.1 RDF Schema

The RDF Schema (RDFS) [4] is a semantic extension of RDF for defining groups of related resources and their relationships. RDFS provides a vocabulary of URIrefs starting with `http://www.w3.org/2000/01/rdf-schema\#` and shortened as `rdfs:`. RDFS URIrefs can be used to define a hierarchy of classes (e.g. `rdfs:subClassOf`), a hierarchy of properties (e.g. `rdfs:subPropertyOf`) and how classes and properties are intended to be used together (e.g. `rdfs:range, rdfs:domain`). As an example the

[3] `http://www.foaf-project.org/`

Figure 2.1: Running example RDF data and graph representation.

statements in Figure 2.1 (c) describe that the class *Employee* (`inetria:Employee`) is a subclass of *Person* (`foaf:Person`). The statements also describe that the property *works at* (`inetria:works_at`) connects instances of type `inetria:Employee` (subject) to instances of type `inetria:Company` (object).

The Semantic Web aims to represent knowledge about resources in a machine readable format that automated agents can understand. Knowledge about a particular domain of interest is described as an *ontology*, which is a machine readable specification with a formally defined meaning [36]. An ontology enables different agents to agree on a common understanding of certain terms, for example *Person* or *Employee* in Figure 2.1. As noted, RDFS provides the capability to specify background knowledge such as `rdfs:subClassOf` and `rdfs:subPropertyOf` about terms, this renders RDFS a knowledge representation language or ontology language. Having said this, RDFS has its limitations as an ontology language, for example it is not possible to express negated statements [36]. The following section provides brief background on the concept of ontologies and the more expressive Web Ontology Language (OWL) [37].

2.1.2 The Semantic Web and Ontologies

In philosophy, an Ontology is a systematic account of Existence. In computer science, Gruber [38] defines ontology as "an explicit specification of a conceptualisation" and conceptualisation as an abstract view of the world represented for a particular purpose. Guarino [39] on the other hand, argues that this definition of conceptualisation relies on an extensional notion and alternatively suggests an intensional definition of conceptualisation. Guarino defines conceptualisation as "an intensional semantic structure that encodes the implicit rules constraining the structure of the a piece of reality" and ontology as "a logical theory which gives an explicit, partial account of

a conceptualisation". In terms of the Semantic Web, an ontology is a collection of definitions and concepts such that all the agents interpret the concepts with regard to the same ontology and hence have a shared understanding of these concepts [40].

2.1.2.1 Description Logics

Existing work in the Artificial Intelligence community have explored the use of formal ontologies in knowledge engineering [41, 42]. For the Semantic Web, a family of knowledge representation languages known as Description Logics (DLs) [43, 44] are used to formally represent the knowledge in ontologies [40]. In Description Logic (DL) classes are called *concepts*, properties or predicates are called *roles* and objects are called *individuals*. The building blocks of DL knowledge bases are called axioms, these are logical statements relating to concepts or roles.

The expressivity of DL is represented as labels for example $\mathcal{ALC}$ (Attribute Language with Complement) is a family of DL that supports all class expressions such as the fact two classes are equivalent or an individual is a subset of particular class or the fact that a role connects two expressions. For example, let A be an atomic class name, R be an abstract role, $\top$ refer to a class that contains all objects (logical equivalence of true), $\bot$ refers to an empty class (logical equivalence of false), $\neg \cap \cup$ refer to class constructor axioms of negation, union and intersection respectively and $\forall\exists$ the universal and existential quantifiers that refer to property restrictions. The class expression C, D in $\mathcal{ALC}$ DL can be constructed as follows [36]:

$$Axiom ::= C \subseteq D | C(A) | R(A, A)$$

$$C, D ::= A | \top | \bot | \neg C | C \cap D | C \cup D | \forall R.C | \exists R.C$$

One of the most expressive DLs in the literature is $\mathcal{SROIQ}$ [46], which is a superset

of $\mathcal{ALC}$. Generally DL axioms can be divided into three categories [45]:

- Assertional or instance data (ABox), which captures knowledge about named individuals i.e. concept assertion such as $smithj \subseteq Employee$, or role assertions between named individuals such as $worksAt(smithj, inetria)$.
- Terminological or schema data (TBox), which describe concepts relationships such as concept inclusion for example $Employee \subseteq Person$ or concept equivalence for example $Person \equiv Human$.
- Relationship between Roles (RBox), which describe concepts such as role inclusion and rule equivalence.

2.1.2.2 Web Ontology Language (OWL)

As discussed previously, as an ontology language RDFS expressivity is limited, for this reason the W3C proposed the Web Ontology Language (OWL) [37, 50]. OWL has two semantics, the RDF-based semantics and the direct semantics, which relate OWL axioms to DL [52]. The first version of OWL, OWL 1 [37] has three sublanguages OWL Full, OWL DL and OWL Lite. The computational complexity [47] for OWL Lite is ExpTime, for OWL DL is NExpTime, and whilst both are decidable, OWL Full is undecidable. These complexities have led many researchers to propose a subset of OWL 1 that can be processed in Polynomial Time (PTime). For example the widely used OWL Horst [48] semantics and Description Logic Programs (DLP) [49].
OWL 2 was proposed to address the computational complexity issues with OWL 1 sublanguages. The W3C proposed three profiles of OWL 2 that can be processed in PTime, namely OWL EL, OWL RL and OWL QL [50]. OWL EL — which belongs to the $\mathcal{EL}^{++}$ [51] family of DLs — is mainly used in biomedical ontologies with very large number of TBox triples such as classes and/or properties. On the other hand, OWL

RL is proposed as the preferred approach for representing Web ontologies that contain very large of instance data (ABox). Finally OWL QL provides database applications with an ontological data access layer [52].

2.1.3 RDFS Entailment Rules

RDF statements also have a formal meaning that determines the conclusions or entailments a software application can draw from a particular RDF graph. The RDFS entailment rules [7] can be applied repeatedly to a set of triples (*graph*) to infer new triples. Other entailment rules related to OWL, include OWL Horst [48] and OWL 2 [50] semantics. The process of applying a set of rules repeatedly to a set of triples, is known as forward chaining or reasoning and continues until no further new triples are inferred, this process is also referred to as *materialisation* [36] in the literature. At this point the closure of the RDF graph is reached under the RDFS semantics and the reasoning process stops.

As an example the rules in Tables 2.1 and 2.2 have a body and a head. The forward reasoning process will add the triple in the head column if the RDF graph contains the triples in the body column, this added triple is usually referred to as an inferred triple. A dataset that contains both the initial triples and all the possible inferred triples is referred to as *materialised* dataset. Some of the rules might infer the same knowledge as others resulting in duplicate statements being added to the RDF graph. This forward reasoning process can be illustrated using the example in Figure 2.1. By applying rule *rdfs9* to the statements in Figure 2.1 (c) and (d), it can be seen that following two statements match the rule's body:

1. `inetria:Employee` **`rdfs:subClassOf`** `foaf:Person`
2. `employee:smithj` **`rdf:type`** `inetria:Employee`

Table 2.1: Schema-Instance (SI) entailment rules of the *ρdf* fragment of RDFS, (*sc = subClassOf, sp = subPropertyOf, dom = domain*)

RDFS	**Body**		**Head**
Name	**Schema Triple**	**Instance Triple**	
rdfs2	?p, rdfs:dom, ?c	?x, ?p, ?y	?x, rdf:type, ?c
rdfs3	?p, rdfs:range, ?c	?x, ?p, ?y	?y, rdf:type, ?c
rdfs7	?p1, rdfs:sp, ?p2	?x, ?p1, ?y	?x, ?p2, ?y
rdfs9	?c1, rdfs:sc, ?c2	?x, rdf:type, ?c1	?x, rdf:type, ?c2

Table 2.2: Schema entailment rules of the *ρdf* fragment of RDFS (*sc = subClassOf, sp = subPropertyOf*)

RDFS	**Body**		**Head**
Name	**Schema Triple**	**Schema Triple**	
rdfs5	?p1, rdfs:sp, ?p2	?p2, rdfs:sp, ?p3	?p1, rdfs:sp, ?p3
rdfs11	?c1, rdfs:sc, ?c2	?c2, rdfs:sc, ?c3	?c1, rdfs:sc, ?c3

The reasoning process will then add the triple in the rule's head to the RDF graph, which in this case is (`employee:smithj` **`rdf:type`** `foaf:Person`). If rule *rdf3* is applied to the same statements, it can be seen that the following two statements match the rule's body:

1. `inetria:works_at` **`rdfs:range`** `inetria:Company`
2. `employee:smithj` **`inetria:works_at`** `business:inetria`

The reasoning process will infer the following statement (`business:inetria` **`rdf:type`** `inetria:Company`). Then by applying rule *rdf9* to this inferred statement and the following statement (`inetria:Company` **`rdfs:subClassOf`** `foaf:Organization`), a new statement (`business:inetria` **`rdf:type`** `foaf:Organization`) can be inferred. Once all the implicit knowledge in an RDF dataset has been inferred, the dataset can be queried by using the Protocol and RDF Query Language (SPARQL) [11].

In addition to forward reasoning, backward reasoning or chaining can also be used to infer new knowledge in RDF datasets. However, backward reasoning is only conducted at query time rather than upfront as with forward reasoning. Forward reasoning has

the advantage of fast queries as no reasoning is required at that point and all the inferred triples are included in the dataset. The disadvantage with this approach is when the knowledge base is updated regularly, in this case the closure needs to be recomputed each time which can be computationally expensive. Backward reasoning on the other hand, starts from the query then builds a knowledge graph for the answer, the advantage of this approach is that the knowledge base can be updated at anytime, the disadvantage is that query answering can take a long time as reasoning is done at query time.

2.1.3.1 A Little Semantics Goes a Long Way

As seen from the previous examples new knowledge can be deduced through reasoning, which means, it is not required to define all the implicit knowledge such as *"iNetria is an Organisation"* upfront. Basic facts can be represented then software applications can be used to deduce further knowledge through reasoning. This feature is best expressed by the so called Hendler Hypothesis [53] that:

> "A little semantics goes a long way."

2.1.4 Minimal and Efficient Subset of RDFS

Some of the RDF and RDFS entailment rules [7] serve the purpose of reasoning about the structure of RDF itself rather than about the data it describes [54]. For example single antecedent rules (rules with one triple in the body), container-management properties and RDF axiomatic triples are usually ignored in other published work [15, 16, 19, 55]. This is because they are easy to implement and are less frequently used. A minimal and efficient fragment of the RDFS entailment rules known as the *ρdf* fragment has been formalised in [54]. The work done in this thesis utilises this

fragment as summarised in Tables 2.1 and 2.2. Below, some of the definitions that will be used throughout the rest of this thesis are introduced:

Definition 2.1.1. ***A schema triple****, is a triple that contains in its predicate one of the RDFS vocabulary terms such as* `rdfs:SubClassOf`, `rdfs:SubPropertyOf`, `rdfs:domain`, `rdfs:range`, *etc., Schema data is a set of schema triples such as the triples shown in Figure 2.1 (c).*

Definition 2.1.2. ***An instance triple****, is a triple that is not a schema triple. Instance data is a set of instance triples such as the triples shown in Figure 2.1 (d).*

Definition 2.1.3. ***A schema-instance (SI) rule****, is a rule that contains in its body one schema triple and one instance triple and contain in its head an instance triple. The RDFS rules in Table 2.1 are SI rules.*

Definition 2.1.4. ***A schema rule****, is a rule that contains schema triples both in its body and head. The RDFS rules in Table 2.2 are schema rules.*

Definition 2.1.5. ***A term****, refers to part of a triple such as the subject, the predicate or the object.*

2.1.5 RDF Triple Stores

RDF datasets in the form of triples are stored and managed in databases usually referred to as triple stores. Triple stores are databases optimised for the storage, processing and querying of triples with the ability to perform forward reasoning under any of the aforementioned entailment rules. Moreover, these triple stores can also support the querying of stored triples using the SPARQL query language. In addition to reasoning and SPARQL support, these stores can also provide RDF management capabilities such the ability to update the knowledge base by adding, modifying or

removing triples. Commercial implementations of large triple stores [14] include, BigOWLIM [56], Virtuoso [57], AllegroGraph and Oracle 11g [58].

2.2 Distributed Semantic Web Reasoning

The current explosive growth of Web data has subsequently led to similar growth in the Semantic Web RDF data. Consequently, recent work on large scale RDFS reasoning has primarily focused on distributed and parallel algorithms utilising peer to peer networks and dedicated computing clusters. The following sections provide a brief survey of related work in the distributed Semantic Web reasoning and data management.

2.2.1 Peer to Peer Networks

Distributed RDF and OWL reasoning on Peer to Peer (P2P) networks has mainly focused on the use of Distributed Hash Tables (DHT) [22, 59, 60]. DTHs provide distributed storage capability for key-value pairs in P2P networks where each node is responsible for a number of keys. The mapping between nodes and keys is done using a standard hash function for both node IP addresses and keys. This mapping is usually referred to as *term based partitioning*, because it is based on fixed mappings from triple terms to nodes. For example a node would be responsible for any triples with the predicate `rdf:type`, another node might be responsible for `rdfs:subPropertyOf`. Each node keeps a lookup table for its successor node so when a request is received for a key that is larger than the one handled by this node, it is forwarded to that nearest successor. This process continues until the key reaches the node responsible for it. Each node supports a Get(Key) and Put(Key, Value) operations on its own database and lookup is performed in $O(\log N)$ where N is the number of nodes.

2.2.1.1 Forward Reasoning on Top of DHTs

Fang et al. [15] presented a system called DORS for forward reasoning on a subset of the OWL entailment rules. They compute the closure of the schema triples using an off the shelf reasoner. The schema triples are then replicated across all the nodes and each node performs reasoning on the instance triples assigned to it. Each node stores these triples in a local relational database and forwards the generated triples to other nodes. This process of triples exchange and reasoning continues until no further triples are sent through the network. Result are reported for datasets of up to 2 million triples of synthetic data using 32 nodes system, and showed scalability problems due to excessive network traffic.

Similar work done by Kaoudi et al. [61, 16] performed forward reasoning on top of DHTs using the *ρdf* fragment of RDFS. They show that forward reasoning on top of DHTs results in excessive network traffic due to redundant triple generation. Their approach also suffers from load balancing issues as some terms are more popular than others. This results in extremely large workloads for some of the nodes compared to others. Although DHTs provide a loosely coupled network where nodes and can join and leave, the approaches summarised previously suffer from load balancing issues.

2.2.1.2 Alternatives to Term Based Partitioning

Kotoulas et al. [19] and Oren et al. [62] have shown that RDF datasets can exhibit a high level of data skew due to the popularity of some terms such as `rdf:type`. This leads to load balancing issues when fixed term partitioning is used as some nodes are overloaded with work compared to others. Oren et al. [62] propose a divide-conquer-swap algorithm called SpeedDate for getting similar terms to cluster around a region of peers rather than at a particular peer. They use an in-memory implementation which speeds up the reasoning process considerably and report results for up to 200M

triples using 64 nodes. However, this approach does not provide a permanent storage mechanism for the generated in-memory data. Also due to the nature of the algorithm used the system has to be stopped manually when no new triples are generated.

Kulahcioglu and Bulut [55] propose a two-step method to provide a schema sensitive, RDFS specific term-based partitioning. They show that, compared to standard term partitioning, they can eliminate non productive partitions accounting up to 45% of the dataset. They analyse each of the *ρdf* entailment rules and identify the term used by the rule for reasoning. This approach avoids using popular terms such *rdfs:type* for DHT partitioning if the term is not used in the reasoning process. Subsequently, they implemented a forward reasoning approach [63] and evaluate it using up to 14M triples. However, no details are provided on loading the data on the cluster and the storage implications of larger datasets.

2.2.2 Grid and Parallel Computing

Work utilising dedicated computer clusters has mainly focused on large scale forward reasoning over datasets that are 1 billion triples or more. In this section a brief review of such related work is provided.

2.2.2.1 MapReduce Based Reasoning

Urbani et al. [64, 65, 20] have focused on utilising the MapReduce programming model [17] for forward reasoning. They presented Web-scale inference under the RDFS and OWL Horst [48] semantics by implementing a system called WebPIE on top of the MapReduce Hadoop framework. They focus on achieving high throughput by reporting the closure of a billion triples of real-world datasets and 100 billion triples of synthetic LUBM [66] dataset. Their approach showed that high throughput can be achieved by utilising only 16 nodes. However, they correlated the reduction of

performance when increasing the number of nodes beyond 16 to platform overhead related to the Hadoop framework.

For RDFS reasoning they find that a naive straightforward implementation suffers from a number of issues including derivation of duplicates, the need for joins with schema triples and fixed point iterations. They provide optimisations for these issues by loading the schema triples in memory due to the low ratio between schema and instance triples. Duplication is avoided by grouping triples by subject then executing a join over a single group. Other optimisations are done for rules with two antecedents which require a joint over parts of the data.

They address the issue of storing large amount of inferred data by using an upfront dictionary encoding [67] based on MapReduce to reduce the data size as will be discussed in Section 2.4. It is worth noting that although this MapReduce approach has achieved very high throughput compared to other systems, it is not without challenges. Performing rule based reasoning using map-reduce jobs require complex and non trivial optimisations, which makes this approach difficult to extend to richer logics such as OWL 2 RL. Additionally the output of this approach is stored in flat files which adds another challenge to provide SPARQL query support.

2.2.2.2 Spark Based Reasoning

To address the issues with the MapReduce batch processing nature, Jagvaral and Park [68] propose an approach based on Spark [69]. Spark is a cluster computing framework that utilises parallel data structures where intermediate results can be stored in memory or on disk. Additionally, Spark enables user to manage their workload partitioning strategy in order to optimise the processing. They show that such an approach is faster than MapReduce by using a cluster of 8 machines each with 8 cores and 93GB of memory. However, no details are provided as to the preprocessing or

the loading process required to get the data on the cluster or provision for long term storage.

2.2.2.3 Authoritative Distributed Reasoning

Hogan et al. [70, 18] have focused on authoritative OWL reasoning over linked data [71] crawled from the Web. They provide an incomplete OWL reasoning by selecting a fragment of OWL then only considering 'authoritative sources' to counteract what they call 'ontology hijacking'. They use two scans over the data, the first scan to separate and filter the schema triples from the instance triples. The schema data is then stored in memory throughout the reasoning process due to its small size compared to the instance data. From the sample data used, it was found that the size of the schema data is less than 2% of the overall statements. The instance data is stored on-disk and accessed through the second scan, during this stage they use on disk sorts and file scans.

Hogan et al. also presented a template rule optimisation for distributed, rule-based reasoning [18]. Instead of simply specifying a list of template rules, they use the schema data to create a generic template rule function that encodes the schema triples themselves into a set of new templated rules. This approach eliminates the need to repeatedly access the schema pattern during the instance reasoning process. The processing is distributed by using Java Remote Method Invocation (RMI) architecture and flooding all nodes with the linear template rules. The only communications required between the nodes is to aggregate the schema data and to created a shared template rule index. They reason over 1.12 billion triples of linked data in 3.35 hours using 8 nodes, inferring 1.58 billion triples.

2.2.2.4 Embarrassingly Parallel Reasoning

Weaver and Hendler [21] presented an "embarrassingly parallel" algorithms for RDF processing. An embarrassingly parallel [72] algorithm indicate that tasks can easily be divided into independent processes which can be executed concurrently without any dependencies or communication between these processes. In this work, two types of supercomputers were used, one with four 16 core AMD Opteron 6272 processors and 512 GB of RAM and an IBM Blue Gene/Q[4], each node has 16 cores at 1.6 GHz and 16 GB of RAM. The system used gigabit ethernet and InfiniBand [73] interconnect and utilised a parallel file system. The data is preprocessed before hand by carrying compression and dictionary encoding as will be discussed in Section 2.4.

For the reasoning process the they utilise up to 128 processes and replicate the schema data on all of them then partition the instance data between the them to execute independently. The reasoning process utilise a Message Passing Interface (MPI) architecture and apply a number of rules in a particular order to derive new triples. This process continues until no further data is inferred. They reported results on the closure of up to 346 million triples of synthetic LUBM datasets in $\approx$ 291 seconds. One of the disadvantages of their approach is that the results of each process are written to separate files leading to duplicates. Additionally, similar to the approach mentioned previously done by Urbani et al. [20], the output is stored in flat files which are yet to be imported into a system that supports querying.

2.2.3 Centralised Systems

Systems based on a single high specification system include BigOWLIM [56], a proprietary large triple store that support the full management of RDF, OWL Horst and a subset of OWL 2 RL ontologies. When the data is loaded BigOWLIM performs

[4] `http://www-03.ibm.com/systems/technicalcomputing/solutions/bluegene/`

rule based forward reasoning upfront, once reasoning is done the data is then stored to provide SPARQL query support. In terms of scalability and resilience BigOWLIM exhibits centralised architecture and vertical scalability. BigOWLIM provides a replication cluster with a number of master and slave nodes. In this architecture the data is replicated across all the nodes and query answering is load balanced between these nodes. The practical limit of BigOWLIM is around 20 billion statements on a node with 64GB RAM, reported results include the reasoning over 12 billion triples of synthetic data using such a node in around 290 hours.

2.3 RDF Data Management on the Cloud

Most of the Semantic Web work that utilises cloud computing services is related to RDF data storage and management [74, 75] rather than performing forward reasoning. Aranda-Andújar et al. have presented AMADA, a platform for storing RDF graphs on the Amazon Web Services (AWS) [27] cloud. They utilised a number of AWS services to index, store and query RDF data. Bugiotti et al. [76] have focused on providing RDF data management in the AWS cloud by utilising SimpleDB [77], a key-value store provided by AWS for small data items. They presented and assessed a number of indexing strategies for their analytical cost models for RDF data stored in SimpleDB for the purpose of providing query answering using out-of-the-box query processor.

Stein and Zacharias [78] presented Stratustore, an RDF store based on the Amazon cloud that stores and indexes triples in SimpleDB and integrates with Jena's API [5] to provide SPARQL query support. They highlighted the limited expressiveness of SimpleDB for this purpose and point to the need for more complex cloud based

[5] `https://jena.apache.org/`

database services. Kritikos et al. [79] have presented a cloud based architecture for the management of geospatial linked data. They utilised an approach based on the autoscaling capabilities provided by Amazon Web Services to scale up virtual machines running the Virtuoso [57] triple store, with the primary focus on providing SPARQL endpoints.

2.4 RDF Data Compression

As noted in Section 2.1, RDF datasets are comprised of triples, each containing three terms — subject, predicate and object — that are represented as strings. When dealing with large datasets, these string representations occupy many bytes and take a large amount of storage space, this is particularly true with datasets in N-Triple format that have long URI references (URIrefs) or literals. Additionally, there is increased network latencies when transferring such data over the network. Although Gzip compression can be used to compress RDF dataset, it is difficult to parse and process these datasets without decompressing them first, which imposes a computation overhead. There is, therefore, a need for a compression mechanism that maintain the semantic of the data, consequently, many large scale reasoners such as BigOWLIM [56] and WebPIE [20] adopt dictionary encoding. Dictionary encoding encodes each of the unique URIs in RDF datasets using numerical identifiers such as integers that only occupy 8 bytes each.

2.4.1 MapReduce-based Dictionary Encoding

In order to perform large scale reasoning, Urbani et al. adopt an upfront dictionary encoding [67] based on MapReduce to reduce the data size. The creation of the dictionary and the encoding of the data was distributed between a number of nodes

running the Hadoop framework. Initially, the most popular terms are sampled and encoded into a dictionary table, since these are small this dictionary held in main memory in each of the nodes. The system then deconstructs the statements and encode each of the terms whilst building up the dictionary table. To avoid clash of IDs, each node assigns IDs from the range of numbers allocated to it. The first 4 bytes of the identifier are used to store the task identifier that processed the term and the last 4 bytes are used as an incremental counter within the task.

A similar approach was adopted for decompression and experimented with a number of settings such the popular-term cache. They report the compression of 1.1 billion triples of the LUBM [66] dataset in 1 hour and 10 minutes, with a 1.9 GB dictionary. Although this approach is scalable both in terms of input and computing nodes, the generated dictionaries take more than an hour to build and are in most cases larger than 1 GB of data. This is a challenge when considering loading the whole dictionary in main memory and imposes an IO overhead as the dictionary file needs to searched in-disk. These large dictionaries are due to the fact that no special considerations are given to the common parts of the URIs such as namespaces, hence including these namespaces numerous times in the dictionary.

2.4.2 Supercomputers-based Dictionary Encoding

Weaver and Hendler [21], use a parallel dictionary encoding approach on the IBM Blue Gene/Q by utilising the IBM General Parallel File System (GPFS). Due to disk quotas restrictions they perform LZO [80] compression on the datasets before the dictionary encoding. LZO is a fast block compression algorithm that enables the compressed data to be split into blocks. This feature is utilised such that processes can directly operate on the compressed data blocks. Subsequently, the compressed file blocks are partitioned equally between the processors, the processors collectively

access the file and starts encoding the data. The encoded data is written in separate output files, one for each processor, additionally, when encoding the data processors communicate with each other using MPI to resolve the numeric identifier for each of the terms. For the dictionary encoding of 1.3 billion triples of the LUBM dataset [81], a reported total runtime of approximately 28 minutes by utilising 64 processors and 50 seconds when utilising 32,768 processors. Both reported runtimes exclude the time required to perform the LZO compression on the datasets. The total size reported for the dictionary is 23 GB and 29.8 GB for the encoded data.

2.4.3 DHT-based Dictionary Encoding

A dictionary encoding based on DHT network is presented by Kaoudi et al. [82] to provide efficient encoding for SPARQL queries. In this approach the dictionary numerical IDs are composed of two parts, the unique peer identifier and a local numerical counter. When new triples are added to the network, they are encoded by the receiving peers and are resent through the network alongside their dictionary entry. As was noted previously, this approach further exacerbates the network congestion issue known with DHTs due to not only the traffic of the triples, but also their encoding.

2.4.4 Centralised Dictionary Encoding

A comparison of RDF compression approaches is provided by Fernández et al. [83]. They compare three approaches, mainly gzip compression, adjacent lists and dictionary encoding. Adjacent lists concentrates the repeatability of some of the RDF statements and achieves high compression rates when the data is further compressed using gzip. They also show that datasets with a large number of URIs that are named sequentially can result in a dictionary that is highly compressible. Additionally, it was shown that dictionary encoding for literals can increase the triple representation spe-

cially when the dataset contains a variety of literals, and hence conclude that literals need finer approaches.

A dictionary approach for the compression of long URIRefs in RDF/XML documents was presented by Lee et al. [84]. The compression is carried out in two stages, firstly the namespace URIs in the document are dictionary encoded using numerical IDs, then any of the URIRefs are encoded by using the URI ID as a reference. Then two dictionaries are created, one for the URIs and another one for the URIRefs. The encoded data is then compressed further by using an XML specific compressor. Although this approach shows compression rates that are up to 39.5% better than Gzip, it is primarily aimed at compacting RDF/XML documents rather than provided an encoding that reduces both the size and enables the data to be processed in compressed format.

2.5 The Challenges

As seen from the previous sections that large scale RDFS reasoning is mainly concerned with the distribution of work amongst a number of nodes or processes. Some of the challenges that existing work try to address are: 1. efficient strategy for workload distribution that deals with data skew and dependencies, 2. a shared storage to accomodate existing and generated data, 3. dictionary encoding to reduce the size of massive datasets.

Additionally, the majority of large scale distributed RDF processing and reasoning utilise high specifications and dedicated computer clusters [19, 20, 21] that require large upfront infrastructure investments. An additional restriction with the approaches surveyed in this section related to the number of processing nodes being used. The number of computing nodes is pre-setup with the required applications and

is fixed throughout the reasoning processes. Most computing nodes that belong to computing clusters also have more or less the same specifications in terms of memory and CPU cores. With this physical computing settings it is difficult to dynamically add or remove computing nodes whilst jobs are in progress. It is also difficult to have computing resources with variable configurations such as memory and CPU cores without expensive upfront infrastructure investments.

In the following sections the dictionary encoding, data storage and workload distribution challenges are briefly discussed.

2.5.1 Dictionary Encoding

The RDF dictionary encoding approaches surveyed in this chapter mainly focused on replacing the long URIRefs to integer identifiers with the aim to reduce the size of the data. However, very minimal considerations are given to the size of the generated dictionaries, which in most cases contain a large level of redundancy due to the shared namespaces and URI path parts. This is clearly demonstrated with the large-scale MapReduce dictionary compression [67], in which case, generated dictionaries are larger than 1 GB for large datasets. Additional the approach followed by [81] uses parallel computations to create the dictionary, which for LUBM was many times larger at 23 GB. This means that the dictionary generation, data encoding and decoding processes take longer to complete as these large dictionaries need to be accessed on disk and can not be held in memory in their entirety.

2.5.2 Data Storage

Most of the distributed reasoning approaches discussed in this chapter require communications between the various nodes to avoid duplication of data and effort. These approaches utilise either, file based approach, local [70, 19] or distributed [20], or a

Distributed Hash Table (DHT) approach [15, 16]. As storage capacity of large scale inferred data usually exceeds the capacity of a single node's hard disk, distributed file storage is usually utilised. This storage needs to have sufficient disk space and network bandwidth to ensure it does not become the bottleneck in the reasoning process. Additionally, as the Semantic Web RDF data is growing continuously there is an ongoing need to add additional storage capacity to existing solutions.

Storage on DHT approaches suffer from considerable network traffic when loading and processing the data, specially in forward reasoning. In addition to this, since the data is partitioned and distributed between the nodes, on a naive DHT implementation if a node fails then the data stored on it is lost. This is not just specific to DHT, but is also applicable to others such as MapReduce where the storage is distributed between the processing nodes. Upfront investment in high bandwidth network infrastructure and storage redundancy is usually required to counter these limitations. An additional limitation with the distributed reasoning approaches discussed here is that they utilise either file or memory based approaches, where the inferred data is yet to be imported in a triple store so that the dataset can be queried.

2.5.3 Workload Distribution

Distributed forward reasoning requires a strategy for partitioning the work between the processing nodes. One of the popular workload partitioning schemes used in DHTs is term-based partitioning. Term-based partitioning is used for distributing the workload not just for reasoning, but also for storage. It ensures that triples sharing a term are located on the same node so inference can take place. As seen in the previous sections that due to data skew, approaches based on term-based partitioning suffer from load balancing issues and do not scale [19].

A number of approaches have been proposed in the literature [55, 19] to address these

issues. In particular the schema sensitive, RDFS specific term partitioning suggested by Kulahcioglu and Bulut [55] is promising due to the fact that it can eliminate the non productive partitions when compared with fixed term partitioning. They propose to create partitions not just for reasoning, but also for storage due to the distributed nature of DHT storage. This thesis builds on this strategy to develop a mechanism for workload partitioning for the reasoning process rather than both reasoning and storage as will be discussed in Chapter 6.

2.6 Summary

This chapter has provided a background on the Resource Description Framework (RDF) and RDF Schema (RDFS). The chapter also presented a review of the literature covering work on distributed RDFS reasoning and RDF data compression. Due to the massive growth of RDF data on the Web to billions of statements, these approaches have primarily focused on using distributed computing for RDF processing, but this is not without challenges. Challenges include data storage, workload distribution and the restriction of fixed computation resources both in terms of number of nodes and the specification of each node. Moreover, existing work on RDF dictionary encoding for large datasets generates large dictionaries which results in a slow encoding and decoding process.

To conclude, existing work on large scale reasoning on RDF Web data has highlighted two important characteristics about this data: 1. the schema data represents only a small fraction of the overall data [70] and can fit in memory during the reasoning process, 2. the schema data should be treated differently to increase performance [85]. This thesis utilises both of these characteristics to address the challenges described in this chapter. The following chapter (Chapter 3) reviews the literature concerning the

cloud computing services that can be used to address the RDF processing challenges presented here.

Chapter 3

The Cloud Computing Paradigm

Chapter 2 has introduced the Resource Description Framework (RDF) and highlighted some of the issues currently facing large scale RDF processing. It was shown that some of these issues are related to computing resources and storage constraints, hence the aim of this research is to address these RDF issues by utilising cloud computing (Figure 1.1). Cloud computing is a computing paradigm that runs on physical hardware and enables users to acquire computing resources on-demand without any upfront investments.

Cloud computing can provide substantial cost savings when compared to physical computer environments that consume energy and incur cost even when the resources are not being used. This is because the cost of running the underlying hardware is managed by the cloud providers and users only pay for the resources they use. Publicly available cloud computing services can be categorised under broad categories, despite the fact that the underlying implementation differs from one provider to another. This chapter provides a review of these services, introduces key concepts and definitions that are utilised to design and develop the CloudEx framework.

The rest of this chapter is organised as follows, Section 3.1 provides a background on

cloud computing definitions. Then, Section 3.2 presents an overview of public cloud deployments and common concepts shared amongst public cloud providers. Followed by Section 3.3, which explains at an abstract level the main cloud services utilised in this research. Then, Section 3.4 provides an overview of the main Google Cloud Platform services used in the CloudEx prototype development and evaluation phases of the research. Subsequently, Section 3.5 categorises and provides a survey of related cloud computing literature and introduces the *cloud-first frameworks* concept. Finally, Section 3.6 concludes this chapter with a summary of the key concepts introduced herein.

3.1 Background

Advances in the virtualisation of computing resources have led to the emergence of the cloud computing paradigm. The National Institute of Standards and Technology (NIST) provides the following definition for cloud computing [23]:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Cloud computing resources are deployed on top of physical hardware (servers, network switches, storage appliances, etc.) hosted in data centres. Because the underlying hardware is always on, it is possible to start and shutdown these virtual computing resources at any time. The ability to instantly provision or release computing resources programmatically using an API is called elasticity [24, 25, 26]. Elasticity

enables a wide range of features, one of which is the ability to script virtual infrastructure deployments. Another feature is the ability to dynamically scale up and down computing resources based on demand. Resources can be scaled horizontally by adding more computing resources or to some extent vertically by utilising computing resources with higher specification.

3.1.1 Cloud Service Models

Public cloud providers, most notably Amazon Web Services [27], Google Cloud Platform [28] and Microsoft Azure [29], utilise their data centres to commercially offer virtualised computing resources. Offered resources broadly fall into one of these service models:

- **Infrastructure as a Service (IaaS)** such as compute, storage and network resources.
- **Platform as a Service (PaaS)** such as databases, servers runtime and messaging.
- **Software as a Service (SaaS)** such as collaboration and productivity suites.

3.1.2 Cloud Benefits

Consumers can benefit from using cloud computing in many ways, generally speaking the benefits can be summarised as follows:

- **On-demand services** - Consumers can provision and scale computing resources automatically when needed without any human interaction with the cloud providers.

- **Pay per use computing** - Consumers only pay for what they use, which is highly cost efficient compared to a physical computing environment that is billed 24/7 even when the resources are not being fully utilised.
- **No upfront infrastructure investment** - With cloud Infrastructure as a Service (IaaS), consumers can instantly acquire the required infrastructure without any expensive upfront investment.

3.2 Public Cloud Deployment Overview

Cloud providers generally keep the details of their internal cloud implementation as proprietary. Nonetheless, there are high level components that are common for most cloud deployments as illustrated in Figure 3.1. Most cloud deployments are organised into geographical regions so that resources are closer to consumers in each region. These regional deployments also serve the purpose of compliance with local or regional data and computer regulations. Each region is comprised of a number of availability zones, each zone is usually a data centre with its own network connection and power supply. Zones are used for increased availability so that major failures in one zone do not impact other zones. Consumers can also deploy their resources in multiple zones for added resilience and availability.

As shown in Figure 3.1 each data centre (zone) contains hardware such as servers, network switches and storage appliances. Each of the physical servers, usually called a host, has a hypervisor application which can create and run virtual machines on the host. Widely used hypervisors for Linux based virtual machines include Xen [86] and KVM [87]. A management component communicates with the hypervisors to facilitate virtual machines scheduling and deployment on each host. If supported, the management component can also facilitate the live migrations of a VM from one

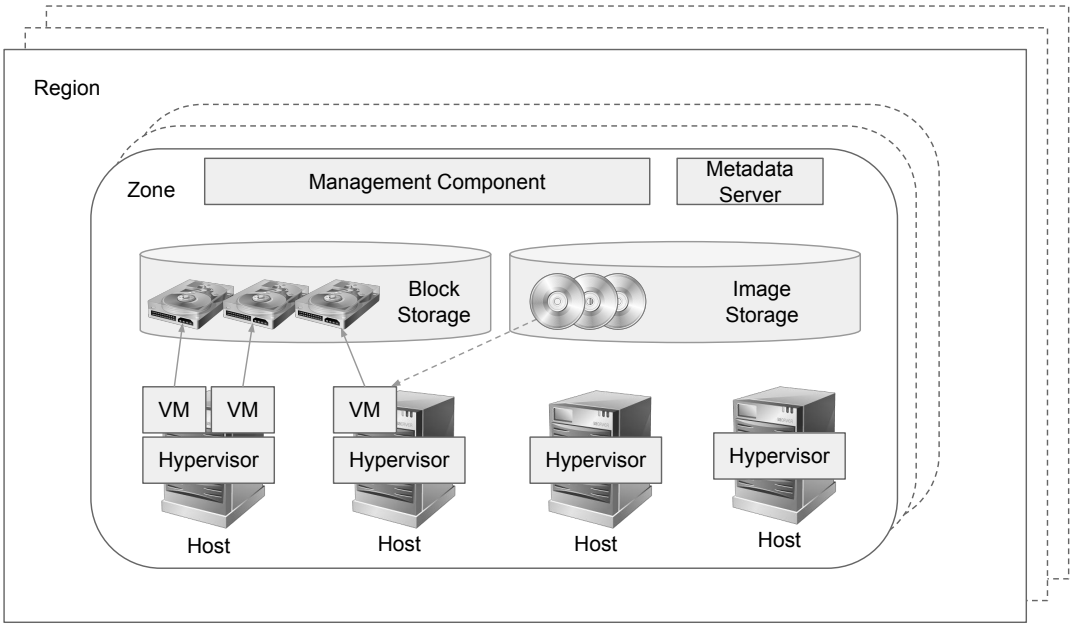


Figure 3.1: Cloud Deployment Overview.

host to another without disrupting the applications running on the VM. Additionally shared storage is also used to store virtual machine images which specify the operating system and applications that are deployed for each VM.

3.3 Cloud Services

Public cloud providers generally categorise their services under broad categories such as compute, storage, databases, Big Data [12], network, etc. The compute category is usually used to refer to services that can be used for computation, for example virtual machines. Storage is used to refer to mass block storage and other types of objects storage. Finally, big data or database categories are used to refer to services that provide fully hosted relational database services, NoSQL or columnar databases [88]. These services still fall under the broad classifications of IaaS, PaaS or SaaS,

depending on the type the service and if it is considered Infrastructure, Platform or Service related. Cloud providers expose authenticated APIs that users can use to manage their cloud resources, the authentication mechanism used differs from one cloud provider to another.

This section provides an overview of the concepts of commonly available cloud services that are utilised in this thesis, namely virtual machines, cloud storage and Big Data services.

3.3.1 Virtual Machines

Virtual Machines (VMs) are created from images (Figure 3.1) and run on physical hosts by the hypervisor component. Unlike physical computers, VMs can be started and terminated programmatically using APIs and are usually billed by the cloud provider for hourly usage. Prominent cloud based compute services include Amazon Elastic Compute Cloud (EC2) [89], Google Compute Engine (GCE) [90] and Microsoft Azure Virtual Machines [91], all do provide APIs for the lifecycle management of cloud VMs. When starting a VM, users must specify the VM image, the VM machine type in terms of CPU cores and RAM, the type of disk storage to use and optionally the network configurations. Some of these concepts are explained in the following sections.

3.3.1.1 VM Image

The VM image contains a boot loader, root file system, Operating System (OS) and the software that will run on the VM. Cloud providers usually provide a set of publicly available images for popular OS distributions, some are free to use and others incur an additional fee depending on the OS license. It is possible to save the current state of the VM's root disk drive as an image. This image can then be used to start new VMs that contain the same software and disk state as the original VM from which

the image was taken. Users can utilise this feature to create their own private images by using the cloud provider images as a starting point. Firstly the public image is used to create a VM, then the OS and applications on the VM are customised and finally a new image is created from the VM.

3.3.1.2 CPU Cores and RAM

Cloud providers specify a catalogue of VM types in terms of CPU Cores, RAM and hourly cost that users can use to create their VMs. A VM CPU core, termed a virtual core, is usually implemented as a single hardware hyper-thread on the physical CPU. This is assuming that the physical CPU does support hyper-threading, which is true for the Intel Sandy Bridge[1], Ivy Bridge[2] and Haswell[3] Xeon processors. For example, a physical CPU with 16 cores and two hyper-threads per core can support a VM with up to 32 virtual cores. Similarly the amount of RAM that a VM can utilise must not exceed the amount of RAM installed on the physical host.

3.3.1.3 Virtual Disk and Snapshots

VMs can use a local virtual storage on the physical host, but this storage is ephemeral and all the data stored on it is lost when the VM is destroyed. Cloud deployments provide persistent block storage that can be used to create VM disk volumes. These disks can be mounted on VMs to provide storage that persists beyond the VMs life-cycle, this is illustrated in Figure.3.1. It is also possible to take snapshots of a VM disk for backup purposes, for example both GCE and EC2 provide the ability to capture incremental snapshots of VM disks. These snapshots are similar to incremental backup in that each snapshot only captures the new changes since the last snapshot.

[1] `http://ark.intel.com/products/codename/29900/Sandy-Bridge#@Server`
[2] `http://ark.intel.com/products/codename/29902/Ivy-Bridge#@Server`
[3] `http://ark.intel.com/products/codename/42174/Haswell#@Server`

3.3.1.4 VM Metadata

Once a VM is created and started, applications running on the VM can query a metadata server for the configuration details of the VM. The details supplied by the metadata server can include the VM type, image, disk and network configurations, start time, IP address, hostname, etc. Users can also supply custom metadata in the form of key-value pairs when creating a VM. These custom values can be read by applications running on the VM in order to make decisions about their processing or lifecycle. This feature can also be used to create similar VMs with applications that use the user supplied metadata as input. For example, based on this metadata applications can configure themselves differently or execute different tasks.

3.3.2 Cloud Storage

Cloud storage generally refers to object based storage rather than block storage used for virtual machines disks. Cloud storage offers a viable replacement to in-premise disk based storage because there are no theoretical constraints on the size of the data that can be stored. Other features of cloud storage include high availability of stored objects through replication, resumable file download/upload and APIs to manage the stored data. Additionally, other data transfer services are also based on cloud storage such as Content Delivery Network services (CDN) with the ability to stream data.
Users can utilise cloud storage for storing data that will be accessed from other services hosted by the same cloud provider. The transfer of the data hosted in cloud storage can directly benefit from the cloud provider's fast internal network and reduced latencies. This is specially true when the data does not need to be transferred over the Internet. Popular cloud storage services include Amazon Simple Storage Service (S3) [92] and Google Cloud Storage [93]. Both cloud providers enable users

to import the input for many of their other cloud services directly from their cloud storage and save the output back to it.

3.3.3 Big Data Services

The constant growth of structured, semi structure and unstructured Web, scientific and sensory data has resulted in the emerging concept of Big Data. Big Data is data that is too big, too fast, or too hard [12] to process using traditional tools. Despite the fact that the challenge of Big Data has been prevalent for the last few decades [94], the rate of growth experienced today makes it hard to store, let alone analyse this data. The seriousness of this is evident by the Big Data prominence in the peak of the Gartner Technology Hype Cycle for 2013 [95]. The main driver behind the hype surrounding Big Data is its potential value, which can enable organisations and businesses to gain valuable insight, trends or business intelligence. For example, the ability to analyse online advertising clickstreams or transactions history data can highlight crucial information about consumer behaviours, hence enabling businesses to tailor their offerings accordingly.

In response to consumer demands for Big Data services, cloud services such as Amazon Redshift [30] and Google BigQuery [31] provide the ability to store, process and query massive datasets. The two services have different models and are defined differently. For example Amazon Redshift is defined as a "petabyte-scale data warehouse", whilst Google BigQuery is defined as "fully managed, NoOps, low cost analytics database", however both utilise columnar database technology [88]. Similarly, both provide SQL like query languages and APIs for the execution of queries and the management of data.

Columnar database systems, unlike traditional Relational Database Management Systems (RDBMS), store data in columns rather than rows. Columnar databases are able

to achieve fast read capability compared to traditional RDBMS, because the values belonging to the same column are stored in the same volume contiguously, tightly packed and compressed [88]. Using this layout, queries scanning different columns can run in parallel and utilise difference disk volumes and hence achieve better disk I/O compared to traditional RDBMS. Different strategies can be adopted to parallelise and aggregate queries that span more than one column. Unfortunately, this fast read capability comes at the prices of slow or almost impossible update to existing columnar data.

3.4 Google Cloud Platform

This section provides a brief review of the key Google Cloud Platform (GCP) services used in the architecture described in this thesis. GCP is one of the prominent commercial cloud platforms and is relatively new compared to AWS, with the first compute service offered in 2013. Some of its major components include Google Compute Engine, Google Cloud Storage and Google BigQuery. To run services on GCP users need to create top level containers called *projects*. Authorisation for resources access, usage quotas and billing for consumed resources are all defined per project. GCP defines a number of regions and zones, for example a zone "europe-west1-b" specifies zone "b" in the Western Europe Region, users needs to specify the zone on which to deploy their resources.

3.4.1 Google Compute Engine

Google Compute Engine (GCE) provides the ability to create and manage Virtual Machines (VMs), also known as *instances*, on the Google infrastructure. VMs are defined under a project and are created in a particular zone that specifies which

region and data centre to host the VM. At the time of writing (December 2015), GCE provides a number of machine types for launching VMs in terms of CPU and RAM, these are summarised as follows [96]:

- **Standard machine types**, include 6 different CPU and RAM configurations ranging from 1 CPU core and 3.75GB of memory up to 32 CPU cores and 120GB of memory.
- **High-memory machine types**, include 5 different CPU and RAM configurations for application that require higher memory than CPU cores. The configurations range from 2 CPU cores and 13GB of memory up to 32 CPU cores and 208GB of memory.
- **High-CPU machine types**, include 5 different CPU and RAM configurations for application that require higher CPU cores than memory. The configurations range from 2 CPU cores and 1.80GB of memory up to 32 CPU cores and 28.8GB of memory.

3.4.1.1 Compute Engine Metadata

Each VM can query the GCE metadata server for information such as VM ID (instance ID), hostname, custom metadata, and startup scripts. Startup scripts are shell scripts that will be run automatically by the VM when it starts up, these are particularly useful to pre-install any required software on the VM. GCE provides enhanced metadata support [97], compared to other prominent cloud providers. For example, GCE provides the ability to update the metadata of a running VM at any time. It also provides the ability for a VM to be notified when its metadata changes by performing a hanging HTTP GET request. There is limited support for metadata by other cloud providers, for example, at the time of writing Amazon Web Services

provides the ability to set the metadata (User Data) for an instance at creation time only, this can not be changed whilst the instance is running. Microsoft Azure on the other hand only provides the ability to set custom tags in the form of key-value pairs when the VM is created.

3.4.1.2 Compute Engine Pricing

GCE VMs are charged for the time they are running based on their CPU and memory configurations. The VMs are charged a minimum of 10 minutes, after the first 10 minutes VMs are charged in 1 minute increments. Other charges include data transfers to and from VMs, disk and images storage per month. Compute Engine VMs can be authorised to interact with other Google Cloud Platform services by using OAuth2[4] open authentication standard. The cloud services that the VM can access need to be specified when the VM is created. Once the VM is started it can query the metadata server to obtain an OAuth2 *token* that can be used to authenticate API requests to those services.

3.4.1.3 Compute Engine API

The following is a list of the Compute Engine API operations used in the architecture developed in this thesis:

- **Instances.insert**, creates a VM under the specified project and zone.
- **Instances.get**, retrieves the details of a VM in the specified project and zone.
- **Instances.delete**, deletes the specified VM in the specified project and zone.
- **Instances.setMetadata**, sets the metadata for the specified instance to the values provided in the request.

[4] `http://oauth.net/2/`

In addition to these API operations, Compute Engine VMs can also read their own metadata or request an OAuth2 token by requesting a particular metadata server URL. VMs can perform a hanging HTTP GET requests to the metadata server such that they can block and wait for metadata changes.

3.4.2 Google Cloud Storage

Cloud Storage is a cloud based service for storing data in the Google infrastructure. Cloud Storage stores data in buckets, which are created under cloud projects and can exist in a particular Region. Buckets reside in a single Google Cloud Platform namespace and as such their names must be globally unique. Objects, which are individual pieces of data, are stored in buckets and have two components, their data and metadata. The object data is usually contained in a file, the metadata is key-value pairs describing the object data. Data transferred to and from other services in the Google Cloud Platform benefits from fast network access by using Google's internal network. Cloud Storage charges for data transfers and data storage per month [98].

3.4.2.1 Cloud Storage API

The following is a list of the Cloud Storage API operations used in architecture developed in this thesis:

- **Objects.get**, retrieves an object or its metadata from a Cloud Storage bucket.
- **Objects.list**, retrieves a list of objects from a particular Cloud Storage bucket, this operation only retrieves the object metadata.
- **Objects.insert**, stores a new object and its metadata in Cloud Storage.

3.4.3 Google BigQuery

BigQuery is a cloud based service for the interactive analysis of massive datasets. BigQuery is an external implementation [99] of Dremel [32], one of the core Google technologies. Dremel technology is capable of running ad hoc queries, without an index, on massive read-only datasets (over trillion rows) in seconds. Dremel achieves this by combining columnar storage structure with multi-level execution trees. Multi-level execution trees provide the ability to break down the query execution over a number of levels of servers down to thousands of leaf servers. Then, each intermediate server in the trees aggregates partial results on the way back up to the root server. Work by Melnik et al. [32] has shown that most queries on this approach complete within 10 seconds, they have also shown that this approach outperforms MapReduce [17] on the same datasets by an order of magnitude.

BigQuery provides an append-only table structure for storing and analysing data, with the ability to store output data from queries into new tables. The table structure is defined using a schema enumerating the columns and their data types. Data can be uploaded directly to BigQuery or through Google Cloud Storage in either JavaScript Object Notation (JSON) or Comma Separated Values (CSV) formats. The data also needs to be de-normalised into a single table for fast access [94]. BigQuery provides a SQL like query languages and REST APIs for the execution of queries and the management of data.

3.4.3.1 BigQuery SQL

BigQuery uses a variation of the standard SQL SELECT statement for its query language [100]. Each SELECT statement performs a full column(s) scan. The basic format of the SELECT query is summarised as follows:

```
SELECT expr1 [, expr2, ...]
```

```
[FROM table\_name1 [, table\_name2, ...]]
[WHERE condition]
[ORDER BY field1 [DESC|ASC] [, ...]]
[LIMIT n];
```

The SELECT expression can contain literals, field (column) names and aggregate functions. The FROM expression contains table names or a nested SELECT clauses. The table name can contain a table decorator [101] (e.g. `table_name@<time>`) with an absolute or relative time in milliseconds. The decorator instructs BigQuery to only scan a subset of the data that was available in the table during the time range provided, for example `table_name@-3600000`, only scans the snapshot of the table at one hour ago. The WHERE clause can contain multiple conditions that make use of comparison operators (e.g. =, !=, <, >, IN, BETWEEN, etc.) and can be joined by AND and OR boolean operations. BigQuery imposes some restrictions on queries, for example the maximum tables per query is limited to 1,000 and the maximum query length can not exceed 256 KB.

3.4.3.2 BigQuery API

The following is a list of the BigQuery API operations used in architecture developed in this thesis:

- **Jobs.insert** is used to create an asynchronous job, there are three types of jobs *Load*, *Query* and *Export*. *Load* jobs are used to load data into a BigQuery table either directly or through Cloud Storage. Data can be loaded into a new table or appended to an existing table. *Query* jobs are used to run a query. *Export* jobs are used to export query results to Cloud Storage files.
- **Jobs.get**, retrieves the status of a particular asynchronous job. A job status

will indicate if the job is *pending, running or done*.

- **Jobs.getQueryResults**, retrieves the results of a query in pages, the maximum page size is 100,000 records.
- **Tabledata.insertAll**, streams the data directly into BigQuery, the maximum rate is 100,000 rows per second.

3.4.3.3 BigQuery pricing

BigQuery provides a number of free operations such as loading, exporting, browsing and copying table data. Query operations are charged based on the size of the data (rounded to the nearest Mbyte) in each column scanned as part of the query. For example the query `SELECT column1 FROM table WHERE column2 = value` will scan both `column1` and `column2` and the total charge is calculated based on the total size of the data in each column. The size of the data stored in BigQuery columns depends on the size of the column data type (e.g. String: 2 bytes + string size, Integer: 8 bytes, Float: 8 bytes, Boolean: 1 byte, Timestamp: 8 bytes or Record: 0 bytes + size of contained fields) [102].

3.5 The Potential of Cloud Computing

Cloud computing has opened up a wide range of possibilities for both organisations and the research community. Cloud computing features such as elasticity and utility billing for computing resources have sparked numerous research efforts and commercial offerings. The related literature surveyed in this section can be categorised into these broad use cases:

- **Migration to the cloud**, the migration of, and interoperability between exist-

ing physical computing clusters or grids and their applications over to the cloud [33, 103, 104, 105, 106, 107, 108].

- **Observing cost and time constraints**, the auto scaling of resources and scheduling of tasks or jobs to meet cost and time deadlines [109, 110, 111].
- **Cloud-first frameworks**, the creation of generic frameworks for data processing, tasks execution, etc., that are entirely based on existing cloud services [112, 113].

Other areas of research concerned with improving the cloud infrastructure and hypervisors are not directly related to the work done as part of this thesis. The following sections provide a brief review of related work in each of these use cases, both in the research and commercial fronts.

3.5.1 Migration to the Cloud

The large majority of work on cloud computing have been concerned with the migration of existing physical computing clusters or grids and their applications over to the cloud. Additionally, the the scientific community have focused on assessing the suitability of cloud computing for High Performance Computing (HPC) [103, 104, 105].

3.5.1.1 High Performance Computing

The research community demand for HPC has led cloud providers such as Amazon, Google and Microsoft to provide support for HPC clusters on their platforms [114, 115, 116]. This is made possible due to improved virtualisation support both at the hardware and software levels and high speed network connectivity protocols such as InfiniBand [73]. For the past few years Amazon Web Services [117] has entered the Top500 list [118] with a virtualised HPC cluster. The Top500 project is concerned

with the 500 top performing HPC clusters around the world and uses the Linpack benchmark [119] to compare performance. The AWS virtualised cluster comprised of 26,496 cores has held a steady rank over the past few years, reaching 81.68% of its theoretical peak performance.

Mauch et al. [104] have shown that the AWS virtualisation performance hit in a single node operation is relatively bearable. However, in a cluster operation using the standard Ethernet, the network latencies are the bottleneck. They show that using high speed interconnect such as InfiniBand can reduce these latencies considerably.

3.5.1.2 Interoperability and Migration of Cluster Frameworks

Other work have focused on improving and extending existing cluster management and job scheduling frameworks to enable migration and interoperability with the cloud [106, 107, 108, 33]. This has mostly taken the form of extending and integrating with existing cluster resource managers such as Torque[5] and job schedulers such as Moab[6] to be able to create cloud based elastic compute clusters.

A large scale elastic environment for scientific computing was presented by Marshall et al. [33]. Their approach integrates with Torque and utilises an elastic resource manager, which consists of three major components, a component to read submitted jobs from a queue, a decision engine that responds to sensor information and a provisioner that interacts with the Cloud provider APIs to launch, terminate and manage instances. VMs are contextualised by exchanging the IP addresses and SSH keys of other VMs in the cluster. The setting up of pre-requisite software on acquired VMs is done using a system integration framework such as Chef[7]. They show that they are able to scale to over 475 Amazon Web Services nodes within 15 minutes.

[5] http://www.adaptivecomputing.com/products/open-source/torque/

[6] http://www.adaptivecomputing.com/products/cloud-products/moab-cloud-suite/

[7] https://wiki.opscode.com/display/chef/Home

An Elastic Cloud Computing Cluster (EC3) tool is developed by Caballer et al. [107]. EC3 creates elastic virtual clusters on top of a number of cloud providers (Amazon EC2, OpenStack and OpenNebula) and integrates with existing resource management systems. The clusters in EC3 are self managed with the ability to scale up or down depending on a predefined policy. Their architecture consists of a launcher that deploys a front-end VM on the cloud, this front-end VM manages the cluster and is able to start, contextualise and stop the VMs in the cluster.

On the commercial front, cloud providers have provided services for managing the deployments of virtualised infrastructure and existing frameworks such as MapReduce on their cloud platforms. For example Amazon CloudFormation [120] and Google Deployment Manager [121] provide users with the ability to define their infrastructure resources deployment as a template detailing the number and specification of VMs and their relationships. Moreover, services such as Amazon Elastic MapReduce [122] and Google Cloud Dataproc [123] enable users to easily deploy and manage MapReduce clusters on their respective compute services (EC2 and GCE).

3.5.2 Observing Cost and Time Constraints

Cloud computing gives users access to an unlimited number of resources subject to physical resources constraints, however there is a cost element involved. Consequently, the majority of research in this area investigated the ability to efficiently schedule specific types of tasks and jobs to meet deadlines and budget constraints. Other areas of research investigates the ability to autoscale cloud resources, mainly VMs to meet specific budget and deadlines.

Grekioti and Shakhlevich [109] focused on revising classical bin-packing algorithms [124, 125] for scheduling Bag-of-Tasks (independent tasks). They propose a new cost optimisation algorithm by combining the existing successful algorithms and optimis-

ing them for handling the computational cost factor. Similarly, [110] developed a statistical cost optimisation approach for scheduling jobs with budget and time constraints between a number of cloud providers with different VM specifications and pricing.

With regard to autoscaling of cloud resources, Mao et al. [111] have focused on cloud autoscaling with budget constraints based on performance and cost. They treated the VM acquisition plan as an optimisation problem and utilised integer programming for solving it. In terms of performance their auto scaling approach enables cloud applications to finish all submitted jobs by the required deadline by acquiring enough VM instances. From a cost perspective they achieve cost reduction by acquiring the appropriate VM instance types that incurs less cost and shut them down when they approach full hour operation.

3.5.3 Cloud-First Frameworks

This thesis introduces the notion *cloud-first frameworks* to denote frameworks that are entirely cloud based and utilise cloud services such as VMs, cloud storage and Big Data services. Such frameworks address particular needs, for example Big Data processing, and provide a Platform as a Service (PaaS) for users to deploy their own applications on the cloud. On the commercial front, Google offers Cloud Dataflow [112], a framework for developing data pipes to process, transform and aggregate large datasets using other GCP services. Amazon on the other hand offers AWS Data Pipelines [113] to process and move data between the different AWS services. Both offerings are transparent to the user, with the cloud provider managing and scaling the underlying infrastructure and users simply paying for using the service.

3.6 Summary

This chapter has provided a review of the cloud computing paradigm including a high level overview of public cloud deployments. The key cloud services utilised in this research were reviewed and an introduction to the Google Cloud Platform was provided. Moreover, this chapter provided a background on some of the related research and commercial work utilising cloud elasticity and utility billing features. Finally, this chapter presented a definition of cloud-first frameworks and explained that they fully utilise cloud elasticity and utility billing. These frameworks utilise not just VMs, but also other cloud services such as storage and Big Data services.

Cloud-first frameworks such as Google Cloud DataFlow and AWS Data Pipelines have two disadvantages 1. they are based on a particular cloud platform and users can not freely move their applications to another cloud provider without considerable changes, 2. the underlying cloud resources being used is not exposed to the user, hence users are not able to specify the number or the specifications of resources to be used in order to optimise their computational usage or reduce cost. The concepts reviewed in this chapter are used in Chapter 5 to address these two disadvantages and to present the architecture of a novel and generic cloud-first framework (CloudEx) that utilises cloud VMs, cloud storage and Big Data services. The approach followed avoids the so-called "lift-and-shift" model for migrating existing approaches to cloud computing, instead this thesis presents a model that is entirely based on cloud computing. The following chapter underpins the design methodology followed in order to develop the CloudEx cloud-first framework and how this framework is then used to develop the ECARF RDF triple store.

Chapter 4

Research Methodology

This chapter focuses on the research methodology followed in this research to answer the research questions summarised in Chapter 1. More specifically, the primary focus is on the *Suggestion* phase of the general methodology of design science research as outlined by [1] (Figure 1.2). The following sections outline the approach followed in order to produce novel designs for both the CloudEx framework and the ECARF triple store.

The rest of this chapter is organised as follows. Section 4.1 provides a brief overview of the methodology used in this research and how it was chosen. Subsequently, section 4.2 outlines the high level design methodology used in order to address the issues with distributed RDF processing. Then Section 4.3 introduces some of the key cloud computing features that influenced the model development stage. Subsequently, Section 4.4 summarises the design methodology used in this research in order to answer the research questions, followed by Section 4.5 which summarises the prototype and evaluation phases of the research. Finally, Section 4.6 concludes this chapter with a summary of the design methodology adopted.

4.1 Design Science Research Methodology

Generally research methodologies can be categorised into quantitative or qualitative, however, sometimes a combined approach is followed. Quantitative research is primarily concerned with numbers and aims to generate data that can be analysed to draw firm conclusions. Examples of quantitative research include experimental research, surveys and questionnaires, etc. On the other hand, qualitative research is designed to help researchers understand some aspect of social life. Examples of this type of research include action research, case studies, etc. In recent years the subject of design science research has gained numerous interest from the information systems community [126, 127]. Design science research changes the state-of-the-world through the introduction of novel artefacts [1].

The primary aim of this research is to develop and evaluate a cloud-based triple store for RDF processing, which requires an iterative process of proposing, designing and evaluating artefacts against related work. Additionally, related work [81, 128, 129] have successfully utilised similar techniques. Based on this, the design science research methodology was chosen for this research, which further facilitates comparison with the findings of related work. This methodology is outlined by Vaishnavi and Kuechler [1] and shown in Figure 4.1. The effort carried out in this research is mapped to this methodology in Figure 1.2. According to Vaishnavi and Kuechler, a typical research effort under this methodology proceeds as follows:

- **Awareness of Problem**. Awareness of an interesting problem in a reference discipline. The output of this phase is a research proposal.
- **Suggestion**. A creative process where new functionality is envisioned based on novel configurations of existing or new elements. The output of this phase is a tentative design.

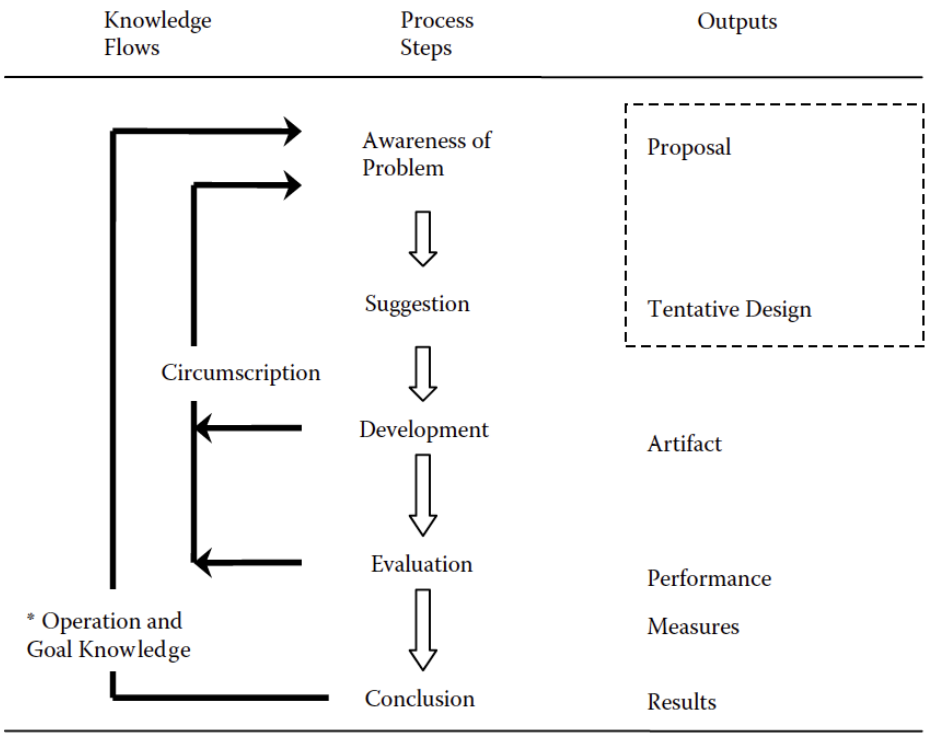


Figure 4.1: The general methodology of design science research (Source: Vaishnavi and Kuechler [1])

- **Development**. The tentative design is further developed and implemented in this phase. The output of this phase is an artefact.
- **Evaluation**. The artefact is evaluated in this phase according to criteria made explicit in the proposal. The output of this phase is performance measures.
- **Conclusion**. This is the finale of a specific research effort, the results are considered good enough and are rewritten up.

The following sections focus on the *Suggestion* phase of this methodology and summarise the creative thought process used to envision new functionality.

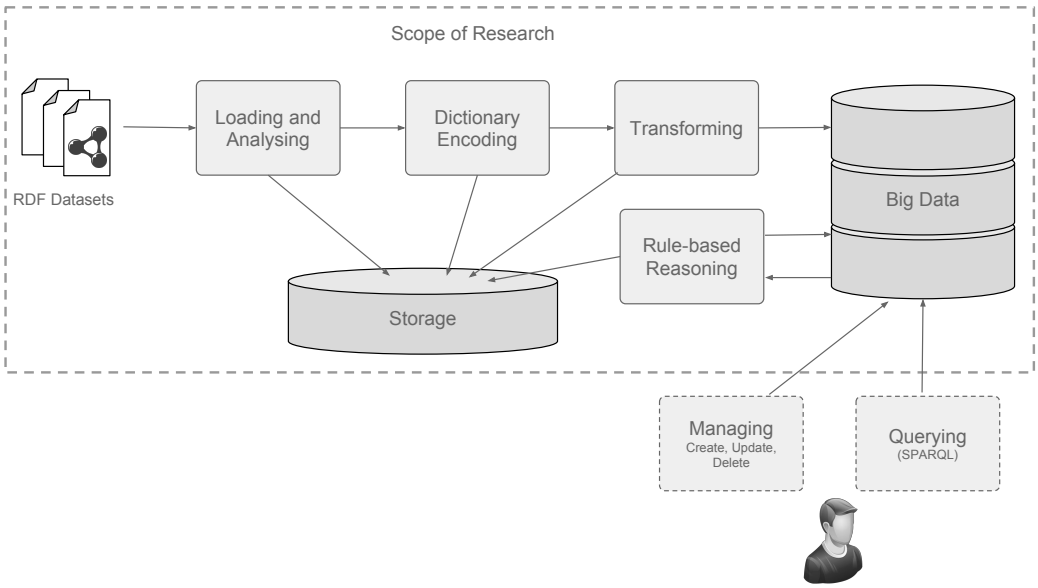


Figure 4.2: Proposed RDF Triple Store Design

4.2 Addressing RDF Processing Issues

A number of issues related to large-scale RDF processing have been identified during the literature review phase of this research, which are summarised Chapter 2. It was identified that the efficient processing and loading of big RDF data benefit from an efficient dictionary implementation, which results in a small dictionary that can readily fit in memory. The ability to hold the entire dictionary in memory is thought to improve the loading times of RDF datasets considerably. Additionally, it was thought that the ability to distribute the processing of these RDF datasets between a large number of computing resources using an efficient workload partitioning strategy will further speed up the processing. Moreover, to address the storage issues it was identified that cloud storage can be used to separate the storage from the computing nodes for both the the dataset files and the final triples. Finally, it was identified that cloud-based big data services can be used for forward RDFS reasoning, storage and query mechanism for both the original and inferred triples. The outcome of this

analysis was the high level triple store design shown in Figure 4.2 and the formulation of the following two research questions:

- Q2. How can an efficient dictionary that fits in-memory be created for encoding RDF URIs from strings to integers?
- Q3. How can cloud-based big data services such as Google BigQuery be used to perform RDFS rule-based reasoning?

4.3 Utilising Cloud Computing

Following the formulation of the aforementioned research questions related to RDF, attention was given to the cloud computing paradigm. It was thought that running an approach based on a fixed computing cluster on cloud computing is unlikely to benefit from cloud elasticity. For example, setting up a MapReduce cluster on a cloud environment to process RDF in an approach similar to [20], will hardly benefit from any cloud specific features. The computing nodes in the cluster are likely to be of the same specifications, preconfigure with the required software, and the number of nodes fixed for the duration of job execution.

4.3.1 Cloud-First Design

The previously mentioned fixed cluster approach is not surprising, given that the MapReduce programming model was first introduced in 2004, almost two years before Amazon Web Services introduced their first cloud IaaS offering (Amazon EC2). With cloud computing there is the potential for computing nodes to be created dynamically, acquired when needed and released when not needed, not only this, but these computing resources can have different CPU cores and memory configurations

to cater for different job or task requirements. Rather than utilising programming models that originated in physical computing settings, there is a need for a cloud-first approach. As a result the following key research question related to cloud computing was formulated:

- Q1. How can a cloud-based system efficiently distribute and process tasks with different computational requirements?

4.3.2 Utilising Cloud Elasticity

When processing tasks on physical computing clusters the constraints are usually related to the number of computing nodes and their specifications. With cloud computing IaaS this is not an issue, the main constraint is related to cost rather than the number of nodes or their specifications. Users can acquire any number of computing nodes, subject to any restrictions imposed by the cloud provider. Users can also choose the specifications of the computing nodes from an inventory of machine types in terms of CPU cores and system memory. This flexibility enables users to utilise a variety of machine types to address particular computational problems. For example a memory intensive task can utilise a computing node with large system memory. Additionally, such a node can utilise multiple CPU cores to provide a level of parallelism by using multiple threads at the application level.

4.3.3 A Divide-Conquer Strategy

The ability to acquire computing resources with various configurations can greatly benefit from dividing computational job into smaller chunks or tasks with each task defining its computational requirements in terms of CPU cores and memory. Ultimately, in such a setting users can acquire a different computing resource for each

task as per its requirements. Consequently, the approach taken in this research is to divide computational jobs into smaller tasks that are independent by following a ”divide-conquer” strategy. The workload for each task can be available as either files in cloud storage or rows in cloud big data services. Using such an approach, processing intensive tasks can be dealt with in an embarrassingly parallel fashion. Broadly speaking, computational tasks can be categorised into the following three categories:

1. processing intensive tasks that can benefit from distribution between computing nodes,

2. memory intensive tasks that require a large amount of system memory and

3. aggregation tasks that are neither processing nor memory intensive.

Using cloud elasticity the appropriate computing resources for the tasks in each category can be acquired when needed and released once the processing is done.

4.4 Model Development

Although the model development stage of the *Suggest* phase started with the design of one model, it became increasingly clear that there were parts concerned with tasks execution on cloud computing and other parts concerned with RDF processing. This realisation is depicted in Figure 4.3 which shows the conceptual model development stage focusing on two models, CloudEx and ECARF. The approach followed was to design any cloud specific task execution models as part of the CloudEx framework, with the intention that this framework is generic and can be used to not only process RDF, but any types of tasks using cloud computing. The CloudEx framework model provided the ability for users to define their own tasks, hence this capability has been

used to define tasks for the ECARF triple store model. Generic RDF processing algorithms were designed as part of the ECARF model.

Furthermore, it was clear that the CloudEx framework is required to be abstract and follows a high level design that is not specific to any particular cloud provider. Subsequently, the model development for CloudEx was divided into two components, a common abstract component and a cloud provider specific component as shown in Figure 4.3. Additionally, a number of requirements have been formulated in order to enable CloudEx to fully utilise cloud elasticity and services, which are summarised in the next section.

4.4.1 CloudEx Framework Requirements

To fully utilise cloud elasticity and services the following requirements can be defined for the CloudEx framework:

- **Cloud-first framework** - It is entirely based on cloud computing services such as cloud IaaS, storage and databases. In this setting, resources can be acquired, released and manipulated programmatically using an API. Data exchange between the various cloud services can also benefit form the cloud provider's network connectivity to reduce latency.

- **Central orchestration** - Central orchestration through a coordinator computing node is needed to coordinate between the various computing resources, manage workload distribution and to release resources when not needed to avoid incurring unnecessary cost.

- **Horizontal and vertical scalability** - The framework enables both horizontal and vertical scalability. Horizontal scalability is the ability to dynamic scale up

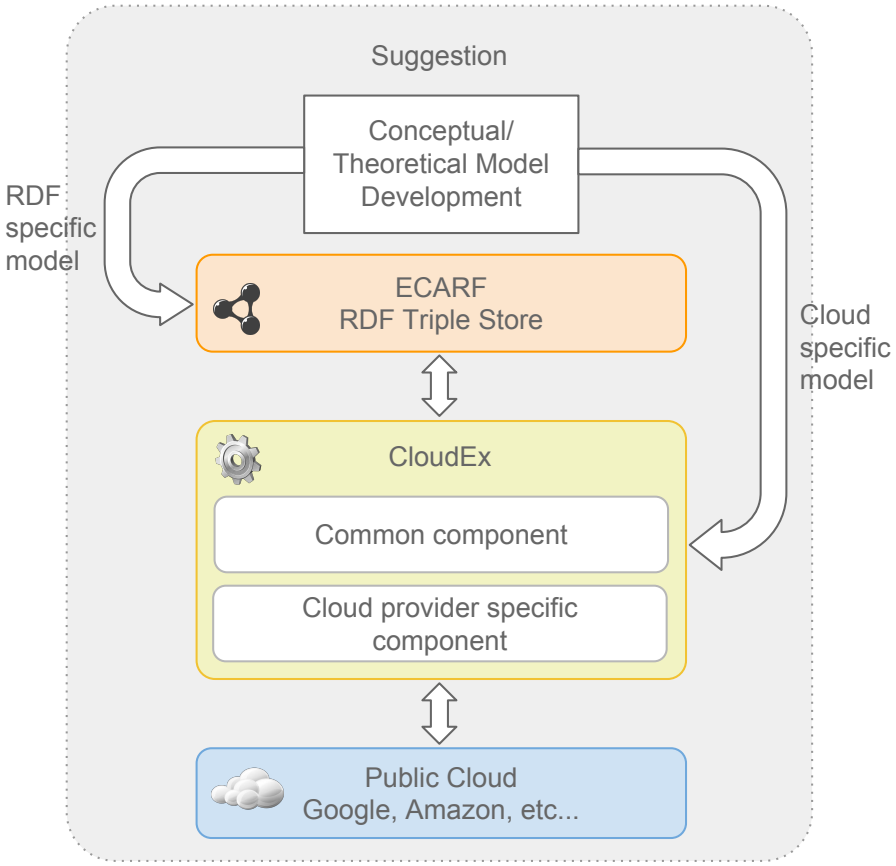


Figure 4.3: Research Design Methodology

or down the number of computing resources during job execution. Vertical Scalability is the ability to acquire computing resources with variable configurations during job execution. These configurations include the number of CPU cores, the amount of system memory and the size and type of disk.

- **Embarrassingly parallel [72] processors** - Computing nodes are only used for processing, they do not store any data locally and do not exchange data with each other by following an "embarrassingly parallel" approach. This approach ensures that nodes can be acquired and released like workers without any interactions with other nodes, with the only interaction with the coordinator. This is achieved by utilising shared cloud storage and moving some of the processing

from computing nodes over to cloud specific services, such as big data services.

These requirements are the main principles behind the CloudEx framework and will be discussed in detail in the next chapter.

4.5 Prototype Development and Evaluation

The overall phases of the research were outlined in Section 1.8 and summarised in Figure 1.2. As was noted, the theoretical model development phase was followed by the prototype development phase which focused on developing working prototypes of both CloudEx and ECARF. Following from this stage experimental evaluation was conducted using both large and small, real-world and synthetic dataset to gather results, check feasibility and make improvements. The results were compared with other related work, additionally, observations were made to identify any areas of the system that were not performing well. Once those areas were identified, improvements were made to the prototype and reflected back to the design, then the evaluation was repeated again. This iterative process continued until it was concluded that the performance of the model was acceptable and showed numerous improvements when compared with related work. At this stage the results were analysed and conclusion were drawn on the findings of the research followed by suggestions for future work.

4.6 Summary

This chapter underpins the design methodology followed in this thesis which is summarised in Figure 4.3. The thought process behind the conceptual model development, the key design decisions and the formulation of the research questions were also summarised. The design methodology outlined in this chapter is used in Chapter 5 for

the design of the CloudEx framework and in Chapter 6 for the design of the ECARF triple store.

Chapter 5

CloudEx, a Cloud First Framework

The cloud computing paradigm made it possible to build dynamic infrastructure that is flexible and elastic, to utilise these features this research designs a **Cloud**-based Task **Ex**ecution framework (CloudEx). Chapter 4 outlined the design methodology for this thesis and introduced the CloudEx framework and its requirements 4.4.1. This chapter covers the first contribution of this thesis and presents the design of CloudEx framework, which answers the following research question:

- Q1. How can a cloud-based system efficiently distribute and process tasks with different computational requirements?

Chapter 3 introduced the various cloud computing concepts including the concept of cloud-first framework, which fully utilise features specific to cloud computing. The chapter presented a number of limitations related to cloud-first frameworks offered by the cloud providers. Namely the difficulty to migrate from one cloud provider to another and the fact that the user has no control on the underlying resources and hence can not optimise their usage. This chapter addresses these challenges and presents a number of generic and novel algorithms that can be used for workload partitioning and distribution between a number of cloud VMs. The CloudEx framework designed and

implemented in this chapter is provided as an open source framework[1] that can be used to efficiently execute jobs composed of tasks with different computational requirements on cloud computing environments. The CloudEx framework is subsequently used in Chapter 6 to design and implement the ECARF RDF triple store.

The rest of this chapter is organised as follows, a description of the high level architecture of CloudEx is provided in Section 5.1. Section 5.2 introduces the CloudEx tasks input and output, then Section 5.3 introduces CloudEx workload partitioning approach and presents a variation of the Bin Packing algorithm. CloudEx job definition is covered in Section 5.4, with Section 5.5 providing a detailed description of the CloudEx job execution. Then, implementation is covered in Section 5.6 and finally the chapter is concluded in Section 5.7 with a summary of the CloudEx framework.

5.1 High Level Architecture

CloudEx is an elastic cloud-first framework for distributed task execution on cloud computing Infrastructure as a Service (IaaS). CloudEx provides generic mechanisms for workload partitioning and distributed tasks execution using cloud Virtual Machines (VMs). The CloudEx high level architecture is shown in Figure 5.1, this architecture is based on three major components, 1. Virtual Machines (VMs), 2. cloud services including databases and storage and 3. the CloudEx framework components. The architecture uses cloud services for shared storage and execution hence eliminating the need for VMs to communicate with each other. This embarrassingly parallel approach ensures each VM can work independently of other VMs to avoid any overhead with data exchange between them. A master VM (coordinator) is used for coordinating a number of other processor VMs or processors for short. The coordi-

[1] `http://cloudex.io`

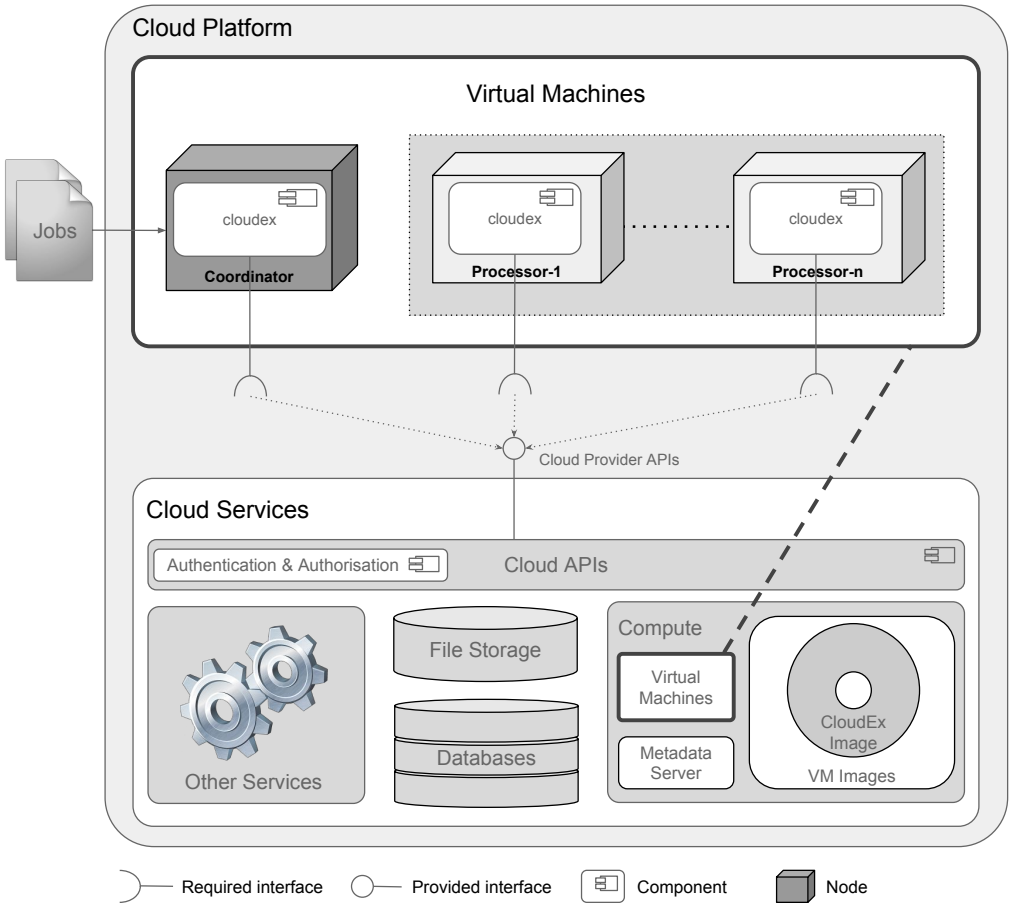


Figure 5.1: CloudEx High Level Architecture.

nator is responsible for partitioning the workload for each task between a number of processors. The coordinator is also responsible for controlling these processors by issuing simple commands in the form of key/value pairs using the *metadata* server (Figure 5.1).

5.1.1 Key CloudEx Definitions

Before continuing, this section introduces some of the key CloudEx definitions that will be used throughout the rest of this thesis. Based on the requirement for CloudEx to provide central orchestration (summarised in Section 4.4.1) cloud VMs are divided

into two categories:

Definition 5.1.1. ***Coordinators*** *(C), are long lived VMs that are left on continuously, these nodes are required to stay alive for as long as the system is running. A coordinator partitions workload using a workload allocation function f.*

Definition 5.1.2. ***Processors*** *(P), are short lived or ephemeral VMs that will be started by a coordinator to perform particular tasks and then be terminated once the tasks are done to avoid incurring unnecessary cost.*

CloudEx work is divided into logical blocks of functionality that is implemented as a subroutine in a computer program, each logical block is called a task, and can be defined as follow:

Definition 5.1.3. ***Tasks*** *(T), are user-defined subroutines that process a number of workload items (files, database rows, etc.) and need to be executed in either a coordinator or a number of processors.*

Tasks that need to be executed by the coordinator are simply referred to as coordinator tasks. Coordinator tasks are mainly responsible for aggregation activities and are neither processing nor memory intensive, as mentioned in Section 4.3.2. Similarly, tasks that need to be executed by processors are referred to as processor tasks. Processor tasks are either processing or memory intensive and require processors to be scaled horizontally or vertically depending on the nature of the task. The ability to scale vertically is achieved by acquiring processors with higher CPU and Memory specifications rather than dynamically scaling up such resources on a particular processor, which is currently a subject of research [130].

A computational job can be divided into CloudEx tasks that are executed separately using a divide-conquer approach. The tasks to be executed by the CloudEx framework

are grouped in a CloudEx job (j), which defines how to orchestrate and execute these tasks.

Definition 5.1.4. ***Job*** *(j), is a definition of CloudEx tasks and how they should be executed. A job also defines some initial data in the form of key/value pairs to be used as input to the task subroutines.*

Each processor task has a workload partitioning function f which is used by the coordinator to partition the workload for the task between a number of processors and can be defined as follows:

Definition 5.1.5. ***Workload partitioning function*** *(f), is the function ($f : W \rightarrow P$) that maps a set of workload items W to a set of processors P.*

The CloudEx framework is implemented as an application component (Figure 5.1) that is run on both the coordinator and processor. This application can be defined as follows:

Definition 5.1.6. ***CloudEx*** *(cx), is an application with a number of subroutines, one for each task and runs on a coordinator or a processor. The application can read and update the processor metadata.*

All processors (P) are created from the same VM image and hence contain the same software. The CloudEx application cx is initially installed on the disk of a template VM and saved as a VM image (vmi).

Definition 5.1.7. ***Image*** *(vmi), is a VM image used to create all the processors. This image contains a bootable root file system and the CloudEx application cx.*

When a processor is created from the image vmi, the CloudEx application cx can do the following based on the metadata supplied to the processor:

- decide which task in *cx* to execute.
- set the task input from the metadata.
- retrieve the workload items assigned to this processor from cloud services (storage, databases, etc.).

5.1.2 The Lifecycle of Coordinators and Processors

The general lifecycle of both the CloudEx coordinators and processors is shown in Figure 5.2. Initially, the coordinator is started and provided with a job to execute. The coordinator will loop through each of the tasks defined in the job, if the task is a coordinator task then the coordinator will execute it. Otherwise for processor tasks, the coordinator will partition the workload between a number of processors using the workload allocation function f. The coordinator then, using image *vmi*, creates and starts the required number of processors, providing each with metadata that contains the name of the task to execute and its input data. At this point the coordinator waits for all processors to finish executing the task.

Once a processor starts up, the application *cx* is run by default. Once running , *cx* reads the processor's metadata, based on this metadata it chooses a task to execute and populate its input data. *cx* then updates the processor metadata status as *BUSY*. When the task is run by the processor it can interact with the various cloud services such as storage and databases to download and process the assigned workload. Once the task is done it uploads its output, if any, to other cloud services. At this point *cx* updates the processor metadata status as *FREE*. The processor then waits for further metadata updates from the coordinator.

Once all the processors have updated their metadata status to *FREE*, the coordinator checks if there are anymore tasks to execute. If all the tasks are done, the coordinator

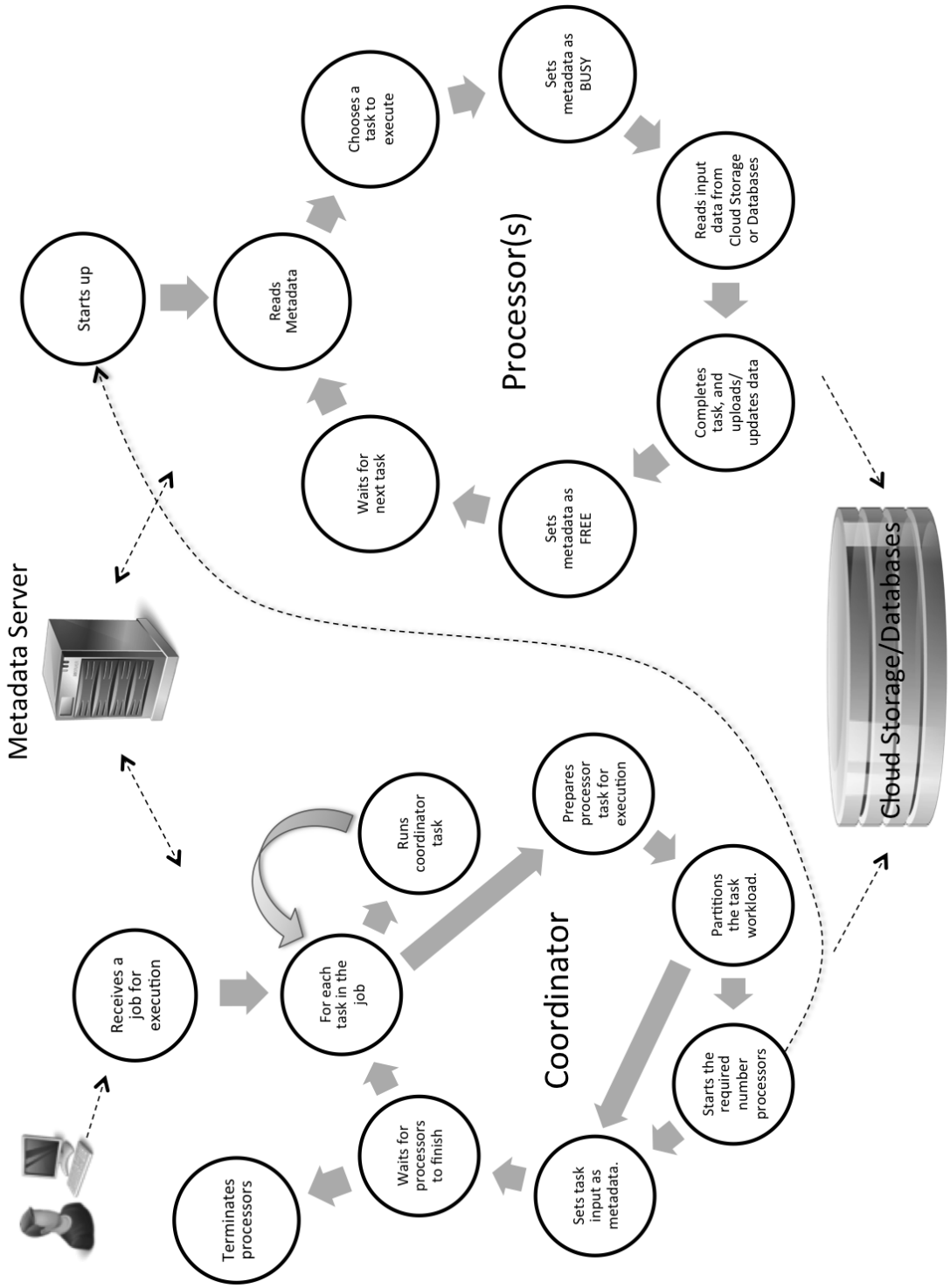


Figure 5.2: CloudEx coordinator and processor lifecycle.

will terminate all the processors to avoid incurring unnecessary cost. If there are more tasks to execute then the coordinator will continue to process them. The processors are reused multiple times during job execution to run all the processor tasks. If a particular task requires more processors than the ones already started then the coordinator will start new ones.

5.1.3 The Tasks Flow

As mentioned in Definition 5.1.3, CloudEx tasks are user defined subroutines that will be run by the CloudEx framework to handle particular workload or data. In addition to distributing task execution between a number of processors, users can also utilise various memory and CPU core configurations. For example, tasks can be made multi-threaded to benefit from multiple CPU cores. This adds a level of parallelism within each processor with the advantage of shared memory access between the various threads. Moreover, tasks that are memory intensive can utilise one or more processors with high memory configuration.

In their simplest terms, CloudEx tasks are defined in a job and executed serially in the order they are defined as shown in Figure 5.3. Tasks such as $Task_1$ and $Task_n$ are executed by the coordinator and hence do not require any distributed processing. These coordinator tasks deal with simple processing and aggregation that is neither computationally nor memory intensive.

Tasks that are computationally intensive such as $Task_j$ shown in Figure 5.3, are distributed between a number of processors. The number of processors used to execute the task is determined by a workload allocation function f. f divides the workload for the task depending on a number of parameters, one of which is the maximum capacity of each processor as will be explained in Section 5.3. The coordinator will delegate the processing of $Task_j$ to the processors and waits for them to finish, once

finished the coordinator then continues with other tasks in the job.

5.2 Dealing with Tasks Input and Output

CloudEx coordinator and processor tasks can accept input and provide output as shown in Figure 5.4. Additionally, every processor task has a partitioning function that can also accept input and provide output. CloudEx provides the ability for users to define how the input for tasks and partitioning functions is populated. It also provides the ability for users to set the output of some tasks and partitioning functions as an input to others. This approach enables users to create a flow of tasks that are dependent on the execution of others and collectively perform a particular job.

5.2.1 The Job Context

Whilst executing a job, the coordinator maintains a *job context*, which is an in-memory collection of key-value pairs. The job context, illustrated in Figure 5.4, plays a central role in data sharing between the various tasks and partitioning functions. The coor-

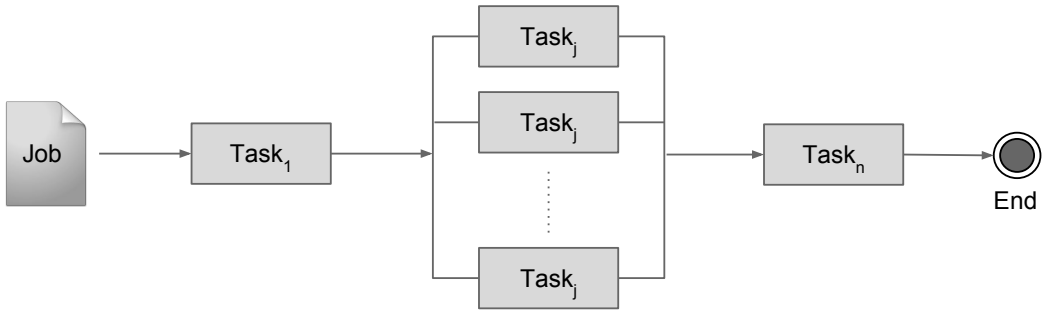


Figure 5.3: CloudEx tasks flow.

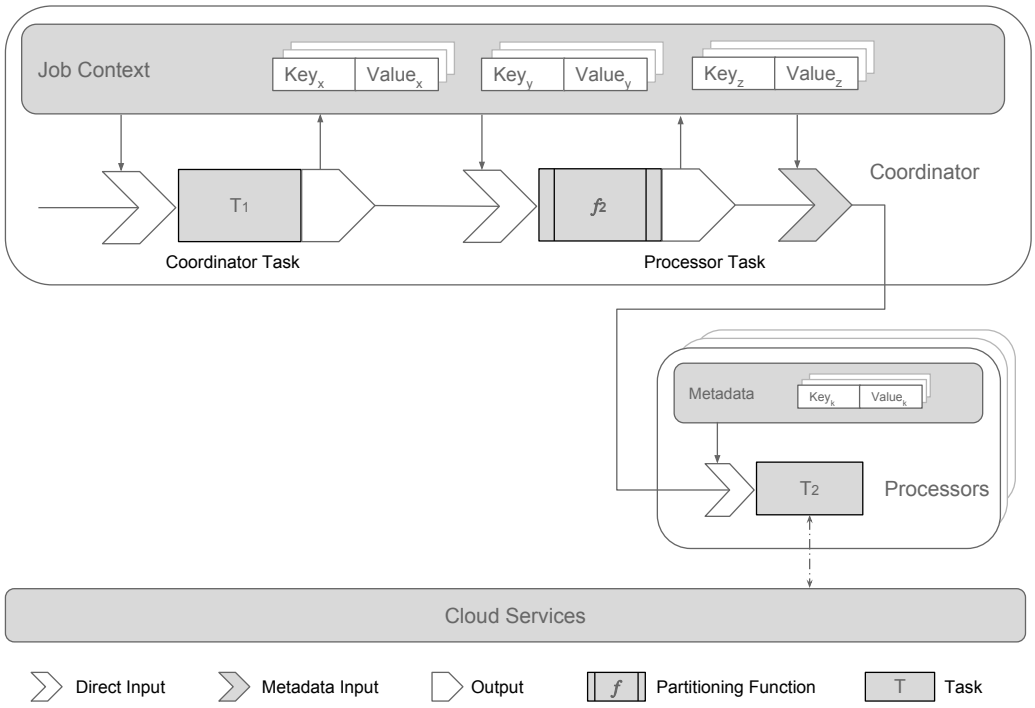


Figure 5.4: CloudEx tasks input and output.

dinator populates the input for tasks and partitioning functions from the job context. Moreover the output of tasks and partitioning functions can also be saved back to it. The coordinator tasks can directly accept input from and provide output to the job context. Processor tasks can also accept input from the job context, however, this input is sent remotely to the processors through the metadata server. Partitioning functions for processor tasks are executed by the coordinator and hence have direct access to the job context. Consequently, these functions can directly accept input from and provide output to the job context. When defining tasks, values in the job context can be referenced as input or output for the tasks, by simply referring to them by their keys. This approach enables users to wire the output of some tasks as an input to others to create dependencies as will be explained in the following section.

5.2.2 Input and Output Resolution

When defining a CloudEx task, its input is defined as either constants or variable names. These variables' names are used as keys to lookup a value from the job context. For example, if a task is defined to have the following key-value pairs as input:

```
{threshold:50,  domain:example.com, table:#key1, file:#key2}
```

and the job context contains the following key-value pairs:

```
{key1:sample-data,  key2:data.txt, key3:some-value, key4...}
```

The coordinator will treat all the values as constants, except those prefixed with `#`, those are treated as variables. In this example, the values for keys `threshold` and `domain` are treated as constants. However, for the other two keys `table` and `file`, the coordinator will remove `#` prefix and will then lookup `key1` and `key2` in the job context. These will resolve to `sample-data` and `data.txt` respectively and the task will be initialised with the following input data:

```
input = {threshold:50,  domain:example.com, table:sample-data,
file:data.txt}
```

The same approach is applied to the input of partitioning functions attached to processor tasks. Additionally, this approach is also used to resolve the input for processor tasks. However, this input is not directly populated by the coordinator, instead it is sent to the processor with the task metadata through the metadata server.

Output of coordinator tasks and partitioning functions can also be specified when defining tasks. The approach for handling the task output is slightly different from the approach described previously for input. For example, if a task has output keys

defined as {`key5, key6`} and the following output is generated when the task finishes executing:

```
output = {key5:value5,  key6:value6, key7:...}
```

Then, only the key-values pairs for the two keys specified in the task definition are added to the job context, i.e. {`key5:value5, key6:value6`}, the rest are ignored. As mentioned previously this variable resolution and input binding approach using the job context enables users to create dependencies between the various tasks. Additionally, it enables users to create tasks that can dynamically determine and set the computational requirements for upcoming tasks in the job.

5.2.3 Handling Input

Each task can be provided with input arguments, which are stored by the coordinator the job context (discussed in Section 5.2.1). These arguments are provided to the task before it starts executing. Coordinator tasks can be directly populated from the job context as illustrated in Figure 5.4 for task T_1.

The partitioning function for a particular processor task can also be provided with input arguments from the job context. This is illustrated in Figure 5.4 for partitioning function f_2, attached to processor task T_2. These arguments are used by the partitioning function to partition the task workload between a number of processors. Unlike coordinator and partitioning function input, which can be directly provided, processor input is provided remotely as metadata input as illustrated in Figure 5.4 for task T_2. The CloudEx application (cx) running on the processor receives requests to execute tasks through the metadata server. cx will then initialise the task and use the relevant metadata key-values pairs to provide the task with its input values.

5.2.4 Handling Output

When finished executing, coordinator tasks can provide output as a collection of key-value pairs, this is illustrated in Figure 5.4 for task T_1. These key-values pairs are added to the job context so they can be used as input to subsequent tasks. When a task is defined the user specifies which key-values pairs from the task output are to be added to the job context.

Partitioning functions for processor tasks can also provide an output to be added to the job context, this is illustrated in Figure 5.4 for partitioning function f_2. This output takes the form of a list of items, the size of this list determines the number of processors to use for the task execution. Each item in the list is then distributed as task input to each one of the processors using the metadata server.

On the grounds that processor tasks are executed remotely in an embarrassingly parallel fashion, they can not directly provide output back to the coordinator. Therefore CloudEx processor tasks do not provide any output arguments. When required to provide output, these tasks save their output directly to one of the cloud services (such as cloud storage), which can later on be accessed directly and aggregated by using a coordinator task.

5.3 Partitioning the Workload

The CloudEx framework provides two mechanisms for users to specify how computationally intensive tasks can be distributed between a number of processors. The number of processors to use for task execution can be determined by using one of these mechanisms:

- a built-in partitioning function f based on a variation of the Bin Packing algorithm [124, 125] or

- a user-defined partitioning function.

The built-in partitioning function f is explained in detail in the following section.

5.3.1 Bin Packing Partitioning

CloudEx provides a built-in workload partitioning function f based on a variation of the Bin Packing algorithm. This function is used to partition the task execution between a number of processors or bins based on the sizes of the workload items W as illustrated in Figure 5.5. Workload items W can be a collection of items which can be resolved from the job context, for example a collection of file names with their sizes. To use built-in function f, one of the following parameters must be provided:

- the number of bins (processors) N to use or
- the bin capacity C and optionally δ (where $0 \leq \delta < 1$) which is the maximum fraction of a bin that needs to be rounded up to a full bin.

The size of each workload item w (where $w \in W$) is a key parameter that needs to be quantified in order to use this approach. For example when processing text files the size could be the total number of lines in each file or it might be the size of the file itself. In particular problem areas, the key parameter used in load balancing might already be known, if not then it can be determined by running experimentation on the relevant data. In Chapter 7 experimentation is done to determine this parameter for each of the tasks that process RDF datasets. CloudEx provides the ability for current tasks to set the partitioning strategy for upcoming tasks dynamically during job execution, enabling users to better adjust their workload partitioning depending on certain outcomes.

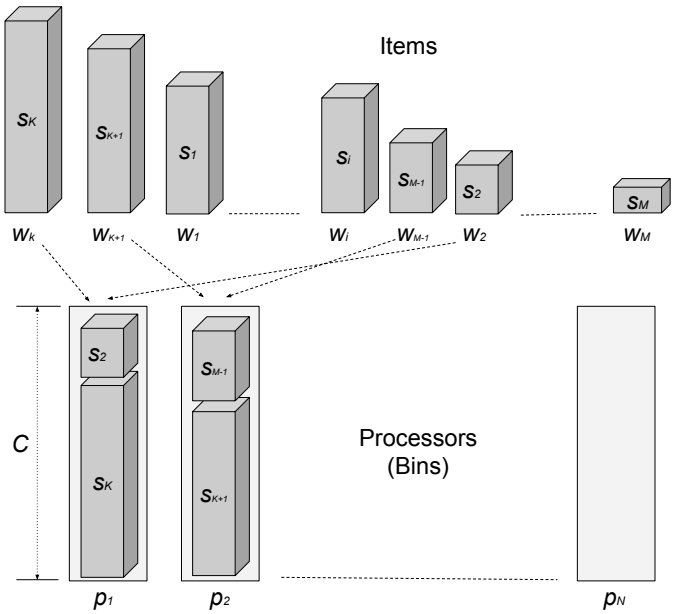


Figure 5.5: Bin packing workload partitioning.

5.3.1.1 Full Bin Strategy

CloudEx uses a Full Bin strategy where all items in W are sorted in decreasing order based on their size as illustrated in Figure 5.5. Given that there are M items in W, for each bin CloudEx iterates through all the items and find the ones that will fully fill the bin up to capacity C. This strategy is not the most optimal in terms of time complexity as it has $O(N * M)$ complexity, compared to $O(n \log n)$ for other bin packing strategies such as First Fit Decreasing and Best Fit Decreasing. However, CloudEx is only dealing with at most a few hundred processors and a few thousand items so the performance impact is negligible.

5.3.1.2 Calculating The Bin Capacity

Assuming there is a total of M workload items $(w_1, w_2, \cdots, w_M)$ and each has a size of $(s_1, s_2, \cdots, s_M)$ with the largest item having a size of s_K such that $(0 < s_i \leq s_K)$ these items are divided into N bins where the capacity of each bin is C. Two cases are

dealt with, the first if the user specifies N, then C is determined as follows, rounding up the results if necessary:

$$C = \frac{1}{N} \sum_{i=1}^{M} s_i \tag{5.1}$$

In this case the number of bins must be fixed, so each bin is filled up to the maximum capacity C. After all the bins are filled any remaining items are equally spread between the bins. This is achieved by sorting the bins in ascending order and the items in descending order then placing the largest items into the smallest bins, this continues until there are no further items to add.

5.3.1.3 Calculating The Number of Bins

The second case is when the user provides both C and δ, in this case N is determined by first calculating the lower bound number of bins N_L as follows:

$$N_L = \frac{1}{C} \sum_{i=1}^{M} s_i \tag{5.2}$$

If N_L has a fraction that is greater than δ then it is rounded up to the nearest integer value. However, if this fraction is smaller than δ then it is discarded and the remaining items after filling up all the bins are spread between the bins as explained previously. If the user does not provide a value for C, or if the provided value is less than the largest item s_K, then the size of the largest item s_K is used.

5.4 Defining Jobs

As mentioned previously the tasks, their input data and processing details for CloudEx are specified using a job definition. The job definition consists of three main parts:

1. Job Data, 2. Virtual Machine Configurations and 3. Tasks Definition. These parts are shown in Figure 5.6 and explained in details in the following sections. An example of a CloudEx job definition in JavaScript Object Notation (JSON) format is provided in Appendix A.2.

5.4.1 Job Data

Job data is a collection of initial arbitrary key-value pairs that can be used as input to the various tasks and partitioning functions. Values denote constants or names of cloud services data containers such as tables or buckets; these values are constraint to String and Numeric data types. When the job is executed, this initial data will be added to the job context so that it can be referenced as input to any of the tasks or partitioning functions.

5.4.2 Virtual Machine Configurations

Virtual Machine Configurations (abbreviated as VMConfig), is a collection of key-value pairs detailing the attributes required for launching the VM. Keys identify common cloud parameters such as virtual machine type, disk type, etc.... Values provide a specific identifier that relates to the cloud provider being used. The VMConfig is used as the base virtual machine settings when creating CloudEx processors. The following keys are used by CloudEx:

- **Zone**: The zone identifier in which the virtual machine should be deployed.
- **VM image**: The identifier of the virtual machine image *vmi* to use, which contains the CloudEx application.
- **VM type**: the type of the virtual machine, the value is a cloud provider specific

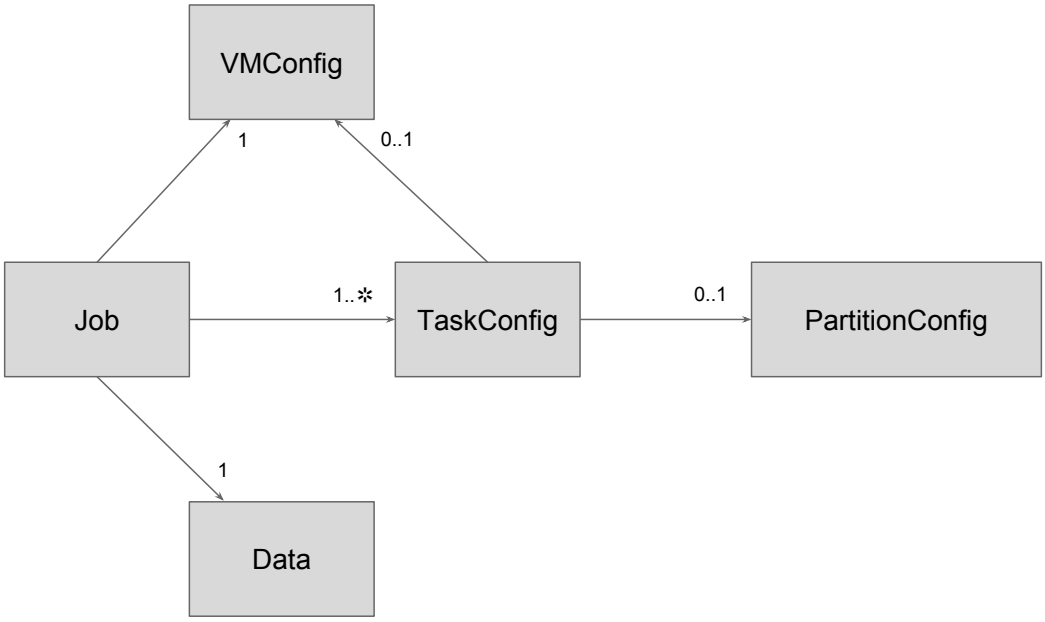


Figure 5.6: CloudEx job entities.

identifier that corresponds to a number of CPU cores and certain amount of main memory.

- **Network**: the identifier of the networking configurations to use for the virtual machine.
- **Disk type**: the type of disk to use, covering options such as magnetic, solid state, ephemeral or persistent disk types.
- **Startup script**: a script to be executed once the virtual machine is started.

5.4.3 Tasks Definition

Tasks definition is a list of task configuration (abbreviated as TaskConfig) for the individual tasks to be executed by CloudEx. Each TaskConfig provides details on how to initialise and process the task. A TaskConfig provide details that are applicable

to either processor tasks, coordinator tasks or both, these details are summarised as follows:

- **Task subroutine (both)**: a reference to the task subroutine to be executed for the task.
- **Input (both)**: the input data for the task specified as a collection of key-value pairs. A value can either be a constant or a reference that will be resolved from the job context.
- **Target (both)**: indicate which component, coordinator or processor that executes the task.
- **Error action (both)**: an action to be taken if the task subroutine exits due to an error. Two error actions are considered, to ignore the error and continue executing other tasks in the job, or to terminate the job execution.
- **Output (coordinator)**: the names of the output of the task subroutine. The output of the task will be added to the job context using this name as the key and the actual output as the value, as explained in Section 5.2.4.
- **VMConfig (processor)**: optionally for processor tasks, a virtual machine configuration (Section 5.4.2) can be specified for the execution of the task. This configuration can be statically specified in the task definition or dynamically specified during the job execution. If omitted the VMConfig provided as part of the job definition will be used for processor tasks.
- **Partitioning configuration (processor)**: the TaskConfig for all processor tasks is required to have a partitioning configuration, abbreviated as PartitionConfig and discussed in the following section.

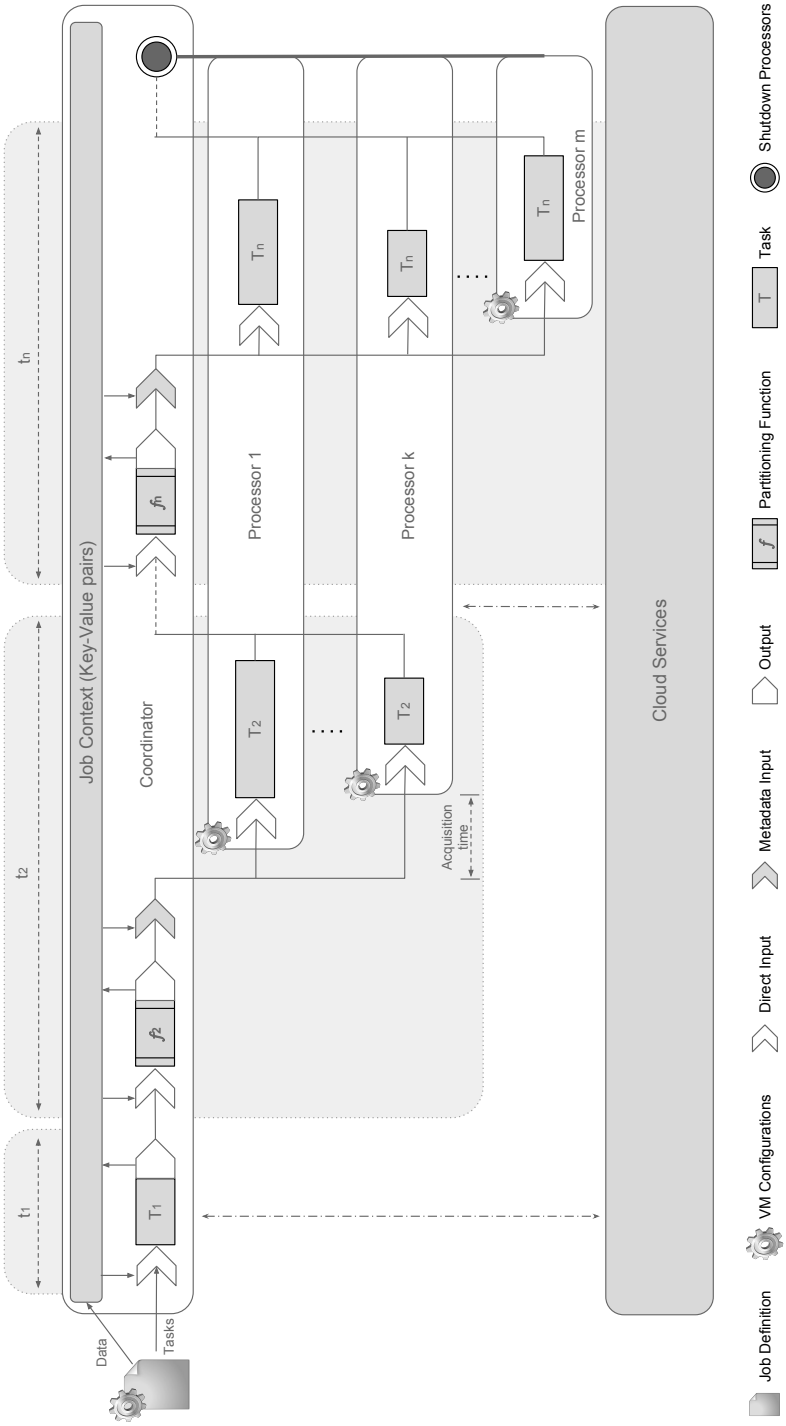


Figure 5.7: CloudEx distributed job execution.

5.4.3.1 Task Partitioning Configuration

The PartitionConfig (Figure 5.6) is required for all processor tasks and contains a reference to the subroutine of partitioning function f and its input data, which can be resolved from the job context. The PartitionConfig also includes output keys to be used when adding the output of f to the job context (Section 5.2.4). This PartitionConfig describes how the workload for the task can be distributed between a number of processors. Partitioning and workload allocation was covered in more details in Section 5.3.

5.5 CloudEx Job Execution in Detail

This section explains in detail the roles of the Coordinator and Processor in CloudEx distributed job execution; a high level overview is provided in Figure 5.7.

5.5.1 Duties of the Coordinator

CloudEx job execution utilises a running Coordinator to sequentially process the tasks defined in the job definition. The activities performed by a coordinator when supplied with a job definition can be summarised as follows:

- Maintain a set of all active processors.
- Execute the coordinator tasks.
- Execute the partitioning function for processor tasks.
- Delegate processor tasks to the active processors, starting new processors when needed.
- Once the job execution is completed, shutdown all the processors.

The coordinator's main algorithm is outlined in Algorithm 1, the processor's main algorithm is outlined in Algorithm 2. The processing for both the coordinator and processors is visually illustrated in Figure 5.7. Initially only the coordinator is started, then when supplied with a job definition, the coordinator adds the job data to the job context. The coordinator then iterates through all the task configurations (TaskConfig) in the job, running and delegating tasks as explained in the following section.

5.5.1.1 Running Coordinator Tasks

When a coordinator task T_1 (Figure 5.7) is encountered the coordinator will lookup the task subroutine. The coordinator then uses the Input keys provided in the TaskConfig to lookup the input values from the job context, then supply them to the task subroutine as explained in Section 5.2.2. The coordinator then runs the task, once done, the output of the task is added to the job context. The total execution time for task T_1 is shown as t_1 in Figure 5.7.

5.5.1.2 Running Processor Tasks

The execution for processor tasks is split between the coordinator and the processors as evident from Algorithms 1 and 2. The coordinator is responsible for executing the partitioning function, for example f_2 in Figure 5.7, to determine the number of processors to use for the task execution. The coordinator is also responsible for populating the input data for the task (T_2) from the job context and delegating the task execution to the processors.

As previously mentioned, the metadata server is used as a mechanism to assign tasks and their input data to the processors. An example of such metadata is shown in Table. 5.1. If the number of processors required for the task execution exceeds the number of active processors, the coordinator will start new ones as needed. Once the

Algorithm 1 CloudEx Coordinator subroutine

```
1: procedure RUN(cs, j)        ▷ cs is the cloud service        ▷ j is the job to execute
2:     P ← {}                                                  ▷ P is a set of processors
3:     ctx ← {}                                                ▷ ctx is the job context
4:     ctx.add(j.data)
5:     tasksConfig ← j.getTasks()
6:     for all tc ∈ tasksConfig do                             ▷ tc is the task configuration
7:         if tc.target = "PROCESSOR" then                     ▷ processor task
8:             vmc ← tc.vmConfig                               ▷ vmc virtual machine configuration
9:             f ← tc.partitionFunction
10:            W ← f.partition(ctx)                            ▷ W is the set of workload items
11:            ctx.add(W)
12:            P_new ← {}
13:            for all w ∈ W do
14:                m ← getTaskMetadata(tc, ctx, w)
15:                p ← P.nextAvailableProcessor()
16:                if p = null then
17:                    p ← cs.startProcessor(vmc, m)
18:                    P_new.add(p)
19:                else
20:                    cs.updateProcessorMetadata(vmc, m)
21:                end if
22:            end for
23:            if P_new ≠ ∅ then
24:                P.add(P_new)
25:            end if
26:            status ← cs.waitForProcessors()                 ▷ block and wait
27:        else                                                ▷ coordinator task
28:            t ← getTask(tc, ctx)  ▷ t is the task to run with its input populated from ctx
29:            o ← t.run(c)                                    ▷ run the task, o is the output
30:            ctx.add(o)
31:        end if
32:    end for
33:    cs.shutdownProcessors(P)
34: end procedure
```

task execution is delegated to the processors, the coordinator will wait for all of them to complete before continuing with the job execution.

The total execution time for task T_2 is shown as t_2 in Figure 5.7. This includes the time required by the coordinator to run function f_2 and the total time taken by the slowest processor to process the task. An example is shown in Figure 5.7, where *Processor 1* takes longer to process the task than *Processor k*. When a processor is started for the first time, there is a slight delay before it is available to process any tasks. This delay, termed *acquisition time* (t_{aq}), is the time required for the processor to be created, started and for the CloudEx application *cx* to start.

Table 5.1: Example CloudEx processor task metadata

Key	**Value**
cloudex-task-class	LoadDataset
user-bucket	dataset
user-files	dataset_file1.nt.gz,dataset_file2.nt.gz,...
user-schema-terms	schema_terms.json
startup-script	/home/cloudex/start_cloudex.sh

5.5.1.3 Error Handling

The coordinator provides two options for handling errors thrown by task subroutines, either to ignore these errors and continue execution or to halt execution and exits providing an error to the user. The option to follow is specified by the user in the TaskConfig for each task. The processor errors are conveyed to the coordinator using the metadata, an example is provided in Table. 5.2

Table 5.2: Example CloudEx processor error metadata

Key	**Value**
cloudex-status	ERROR
cloudex-exception	NetworkAccessError
cloudex-message	No network access

5.5.2 Elastic Processors

As previously discussed in Section 5.4.3, the TaskConfig for processor tasks can provide a VMConfig to use for the task execution. The VMConfig provided for a particular processor task can be different from the one provided at the job level. This approach enables the execution of certain tasks in processors with different virtual machine configurations, such as main memory and CPU. This is particularly useful when dealing with tasks that are memory or both CPU and memory intensive. The coordinator keeps a list of all active processors, their virtual machine configurations, start time and accumulated cost. If a task requires a processor with particular VMConfig that is different from the one provided at the job level, then the coordinator will check the list of active processors for a VMConfig that matches the configuration required for the task. If one is found then it is used, otherwise the coordinator will start a new processor with the required VMConfig. The default action is to shut this processor as soon as the task processing is done, unless the user specifies otherwise in the TaskConfig. This is done to avoid incurring cost for expensive virtual machines that have high memory or CPU cores.

In a push to achieve a high level of elasticity, not only can the VMConfig for a task be specified statically, it can also be driven dynamically from the job context. For example, for a particular processor task, the VMConfig can be specified as a job context key reference that can be resolved by the coordinator at runtime e.g. `#customVMConfig`. This is useful if a coordinator task is used to determine the computational capabilities required for the next processor task. This coordinator task then outputs the VMConfig which will be added to the job context e.g.: {`customVMConfig: cup-32-mem-208`} where `cup-32-mem-208` corresponds to a virtual machine with 32 CPU cores and 208 GB of system memory. The processor task is then delegated to the required number of processors with these configurations.

5.5.3 The Processor Duties

As mentioned previously, processors are short lived virtual machines that are started to process tasks in a job and are shutdown once the job is done. The main processor algorithm is outlined in Algorithm 2, the processor's main subroutine loops indefinitely waiting for tasks. Since processors are only started when a task needs executing, as soon as the processor is started there will be a task metadata available, for example T_2 shown in Figure 5.7. An example of such metadata is shown in Table. 5.1.

Before running a task the processor updates its metadata with a special key-value pair to indicate to the coordinator that it is busy e.g. {`cloudex-status: BUSY`}. Whilst processing, the task subroutine can access the various cloud services to download, process and upload assigned data. The task can also be implemented to use multiple threads to utilise more than one CPU core.

Once the processing of the task is done, the processor updates its metadata to indicate its availability for further work e.g. {`cloudex-status: READY`}. The processor then blocks and waits for metadata updates from the coordinator for further tasks to execute. This lifecycle continues until the processor is explicitly shutdown by the coordinator as shown for task T_n in Figure 5.7.

5.5.3.1 Error Handling

The processor subroutine will capture all errors thrown by the task subroutines, stops processing the task and updates its metadata changing its status to ERROR (Table. 5.2). The processor then blocks and waits for metadata updates from the coordinator.

Algorithm 2 CloudEx Processor subroutine

```
1: procedure RUN(cs, m) ▷ cs is the cloud service ▷ m is the metadata for the processor
2:     status ← null
3:     while true do                                                    ▷ main loop
4:         if status = null then
5:             m.status ← "BUSY"
6:             cs.updateMetadata(m)
7:             t ← getTask(m)        ▷ t is the task to run, with its input populated from
   metadata m
8:             if task ≠ null then
9:                 t.run(cs)                                            ▷ run the task
10:            end if
11:            m.clear()
12:            m.status ← "READY"
13:            cs.updateMetadata(m)
14:        end if
15:        m ← cs.waitForMetadataChange()                               ▷ block and wait
16:        status ← m.status
17:    end while
18: end procedure
```

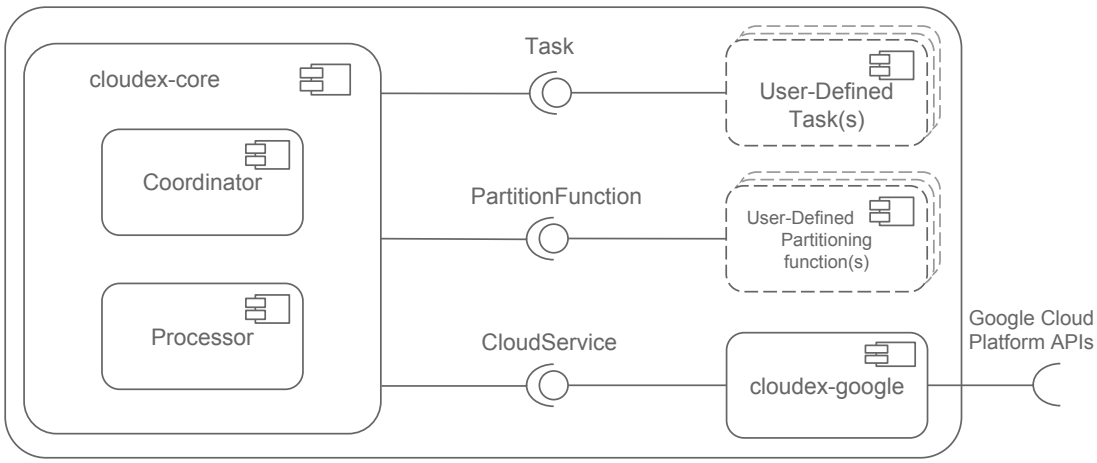


Figure 5.8: CloudEx high level components.

5.6 Implementation

An implementation of the CloudEx framework was created using the Java programming language. This implementation is open source and is publicly available[2]. The

[2] `https://cloudex.io/`

current implementation of the CloudEx framework consists of the high level components shown in Figure 5.8 and summarised as follows:

- **cloudex-core**: is the generic implementation of the processor and coordinator algorithms and provides a *Coordinator* and *Processor* sub components.
- **cloudex-google**: is an implementation of the `CloudService` interface which is specific to the Google Cloud Platform (GCE) [28].
- **user-defined-tasks**: is a component of all the user defined tasks subroutines that the CloudEx framework needs to execute.
- **user-defined-partitioning-functions**: is a component of all the user defined partitioning function subroutines that the user can reference when defining processor tasks. CloudEx includes the built-in Bin Packing function explained in Section 5.3.

5.6.1 cloudex-core Component

The cloudex-core component (Figure 5.8) provides generic implementation of the core CloudEx building blocks described in this chapter. This implementation is generic and can be used with any cloud services that provides APIs for creating and managing virtual machines. This component requires a `CloudService` interface which provides access to the underlying cloud provider services. The following are some of the key operations required by this interface:

- `init()`: initialises the cloud service by performing any required authentication with the cloud provider.
- `getMetaData()`: retrieves the metadata for the current VM or other VMs started by the CloudEx framework.

- `updateMetaData(metaData)`: updates the metadata for the current VM or other VMs started by the CloudEx framework.
- `startInstance(vmConfig)`: starts one or more VM instances.
- `shutdownInstance(vmConfig)`: shuts down one or more VM instances.

Other operations include common operations for accessing cloud based storage and databases. The cloudex-core component also includes a *Coordinator* and *Processor* sub components. The *Coordinator* sub component is run by the Coordinator VM and the *Processor* is run by the Processor VM.

5.6.2 cloudex-google Component

The cloudex-google component is an implementation of the `CloudService` interface required by the cloudex-core component. The cloudex-google component can interact with the various GCE RESTful APIs. The choice to use GCE in this thesis is due to their enhanced metadata support [97], as mentioned in Chapter 3. In order to provide implementation for other cloud providers, the missing features need to be supplimented, in particular metadata support, to match the functionality provided by GCE. This can be implemented as a custom metadata server that runs in the coordinator. The coordinator's IP Address can be provided to the processors when they are created.

5.6.3 User-Defined Tasks

The cloudex-core component provides a `Task` interface for user defined tasks. Users implement this interface to create coordinator and processor task subroutines (Figure 5.8). These subroutines will automatically get an implementation of the `CloudService`

injected at runtime, all input data will also be injected. The task interface has a `run` method that will be run by the cloud-ex framework. In this method users can utilise the `CloudService` to interact with the various cloud services that the framework has authorisation to access. Tasks can also query the number of available CPU cores and fork multiple threads accordingly to provide a level of parallelism within the task execution.

5.6.4 User-Defined Partitioning Functions

In addition to the CloudEx provided partitioning function f, user can define their own partitioning functions. The cloudex-core component provides a `PartitionFunction` interface that users can implement with their own logic to partition the workload between the various processors. The interface defines a `setItems` method, used to set the workload items that need partitioning. It also defines a `partition` method that will be called by the cloudex-core component to perform the actual partitioning. This approach gives users the flexibility to indicate how each processor task should be partitioned for distribution.

5.6.5 VM Image Setup

All CloudEx processors are initialised from the same Virtual Machine image *vmi*, which contains the operating system (OS) and the software required to run the CloudEx framework. In addition to the OS, the minimum required software is the Java Virtual Machine (JVM) and the CloudEx framework. *vmi* is initially created by starting a VM using one of the default images available from the cloud provider. Then all the required software is installed, the VM is then shutdown and an image is created from its disk and saved as *vmi*. Detailed instructions on how to setup the Virtual Machine for CloudEx on the Google Cloud Platform are available on the

CloudEx website[3].

User defined tasks and partitioning functions can either be build as part of the image or installed when the VM starts by using the Startup-Script mechanism (discussed in Chapter. 3). Other software can also be installed using this mechanism, however downloading and setting up software on startup will increase the VM acquisition time t_{aq} (Section 5.5.1.2).

5.6.6 Running the Framework

The coordinator is the minimum component that needs to be running in order to use the CloudEx framework. The coordinator can be a VM deployed on the cloud provider IaaS or a physical computer, as long as the CloudEx framework has the required credentials to authenticate with the cloud provider. To run the coordinator two components are required, the JVM and the CloudEx framework. Users can utilise and extend the CloudEx framework by import the framework libraries into their own applications and frameworks.

5.7 Summary

This chapter have focused on the first contributions of this thesis to answer the research question "*Can a cloud-based system efficiently distribute and process tasks with different computational requirements?*". The chapter introduced CloudEx, a generic and open source cloud-first task execution framework that can be implemented on any cloud provider IaaS. The algorithms presented for CloudEx, address the issues highlighted in Chapter 3 with cloud-first frameworks. This is achieved by providing generic mechanisms that are cloud provider independent for executing tasks on cloud

[3] `http://cloudex.io`

environments. Additional components that enable CloudEx to work with a particular cloud provider can be added, enabling users to easily move their applications from one cloud provider to another. Moreover, a workload partitioning approach based on the bin-packing algorithm is also introduced to efficiently distribute the processing of tasks between a number of processors.

This chapter has outlined the various components of CloudEx including the coordinator and processor virtual machines. An approach based on the bin-packing algorithm was presented for workload partitioning between the various processors. Additionally, CloudEx enables users to use a divide-conquer approach to divide their jobs in multiple tasks and specify how these tasks should be executed. Users can specify not only the number of required processors, but also their types in terms of CPU cores and memory. An implementation for the Google Cloud Platform was also presented. Chapters 6 builds on CloudEx and define the architecture of ECARF, a triple store for RDF dataset processing and forward reasoning on the Cloud. Chapter 7 evaluates this implementation in processing real-world problems such as RDF data processing.

Chapter 6

ECARF, Processing RDF on the Cloud

The explosive growth of RDF datasets to billions of statements have resulted in solutions that utilise high specification hardware, which requires considerable upfront investments. Cloud computing on the other hand has motivated this research to develop solutions that can efficiently process large datasets with billion of statements without any upfront investment. In this regards, Chapter 5 introduced the first contribution of this thesis and outlined the architecture of CloudEx, a cloud-first tasks execution framework. This chapter continues the contribution of this thesis and describes the design of an **E**lastic **C**ost **A**ware **R**easoning **F**ramework (ECARF), a cloud-based RDF triple store implemented as CloudEx tasks. The algorithms described in this chapter addresses the issues with large RDF processing outlined in Chapter 2, namely dictionary encoding, data storage and workload partitioning. Additionally, these algorithms provide answers to the following research questions:

- Q2. How can an efficient dictionary that fits in-memory be created for encoding RDF URIs from strings to integers?

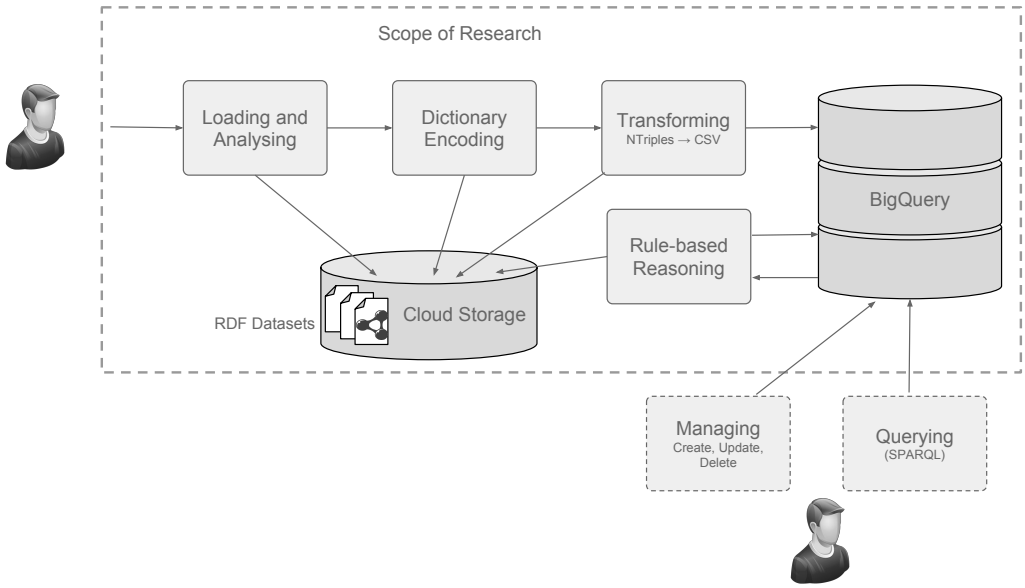


Figure 6.1: ECARF High Level Activities

- Q3. How can cloud-based big data services such as Google BigQuery be used to perform RDFS rule-based reasoning?

Some of the high level activities that can be performed with ECARF are illustrated in Figure 6.1. The activities with solid border such as "Loading and Analysing", "Dictionary Encoding", "Transforming" and "Rule-based Reasoning" are implemented in this thesis. Other activities with dotted border such as "Managing" and "Query" can be implemented to provide full triple store capability, though, these are left as future work. ECARF algorithms that load, process, encode and reason on large RDF datasets will enable applications to easily and efficiently utilise RDF technologies to provide many features such as semantic search, content discovery, etc....

The rest of this chapter is organised as follows, Section 6.1 provides an overview of the ECARF triple store architecture, followed by Section 6.3 which explains the distributed RDFS reasoning approach in details. Then, Section 6.2 presents the dictionary encoding algorithms used by ECARF, followed by Section 6.4 which provides

a detailed walkthrough the architecture. Finally, Section 6.5 concludes this chapter with a summary of the key contributions of ECARF.

6.1 ECARF Overview

The ECARF high level architecture shown in Figure 6.2 is based on the CloudEx's cloudex-core and cloudex-google components. Both the coordinator and processors virtual machines run the cloudex-core `Coordinator` and `Processor` components respectively. These components use the cloudex-core's `CloudService` component to interact with other cloud services. ECARF defines a number of CloudEx jobs and user defined tasks (Section 5.6.3) for RDF datasets processing. Furthermore, ECARF extends the cloudex-core's `CloudService` component with additional API wrappers for the Google BigQuery. Similar to the CloudEx implementation, the ECARF implementation is open source and is publicly[1] available.

The ECARF architecture uses both Cloud Storage and BigQuery for shared storage and BigQuery for execution, hence eliminating the need for processors to communicate with each other. This embarrassingly parallel approach ensures each processor can work independently of other processors to avoid any overhead with data exchange between them. A CloudEx coordinator component is used for partitioning the workload between a number of processors using the CloudEx built-in Bin Packing partitioning described in Section 5.3. Additionally, the coordinator assigns tasks and workload items to the processors by using the *metadata* server as described in Section 5.2. The following sections provide a brief overview of each of the activities explained in this chapter.

[1] `http://ecarf.io`

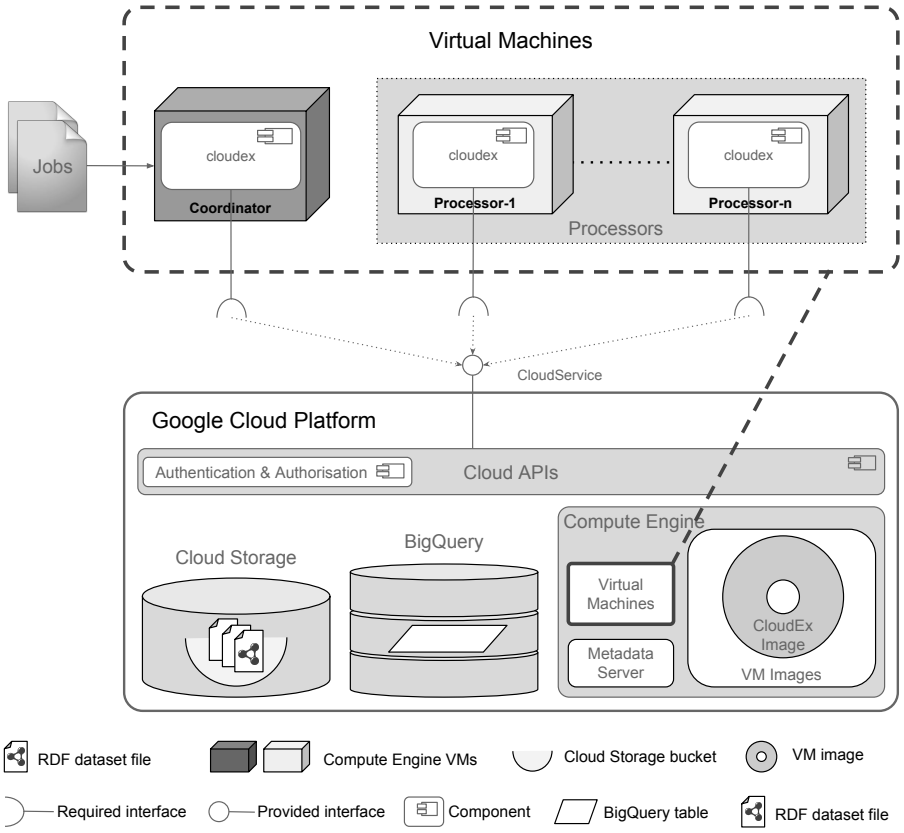


Figure 6.2: ECARF High Level Architecture

6.1.1 Distributed Processing

The input for ECARF is provided as compressed or uncompressed RDF dataset files initially stored in Cloud Storage. The processing of these dataset files is distributed between a number of processors. Each processor works independently to gather statistics about the dataset such as counting the occurrences of particular terms, the results of each processor are then aggregated by the coordinator. Additionally each processor can extract certain statements or terms, or transform the dataset files assigned to it into a different format, for instance the dataset files can be transformed from N-Triples [35] to CSV format — which is supported by BigQuery — and stored back into Cloud Storage. Whilst processing the dataset files, each processor can also perform encoding

or compression on the dataset. The ability to efficiently distribute the processing of RDF dataset files directly impacts the dictionary compression and forward reasoning activities summarised in the next two sections.

6.1.2 Dictionary Encoding

As mentioned in Chapter 2, RDF datasets are comprised of triples, also known as statements, each statement contains three terms — subject, predicate and object — that are represented as strings. When dealing with large datasets, these string representations take a large amount of storage space, this is particularly true with datasets in N-Triple format that have long URI references (URIrefs) or literals (Section 2.1). Additionally, there is increased network latencies when transferring such data over the network. Although the algorithms presented in this chapter deal with datasets that are binary compressed using Gzip, it is difficult to parse and process these datasets without decompressing them first, which imposes a computation penalty. There is, therefore, a need for a compression mechanism that maintain the semantic of the data, which leads to the adoption of dictionary encoding.

During the distributed processing of the dataset files, all the unique terms — blank nodes, literals or URIrefs — in the RDF statements are extracted. Consequently, these unique terms are encoded using integer identifiers then a dictionary is built with these identifiers alongside their relevant terms. By encoding the string data into integers the storage size is reduced, consequently reducing the cost of storage and processing of this data in BigQuery as explained in the next section.

6.1.3 Forward Reasoning

The objective of processing RDF datasets using ECARF is to be able to load such datasets into BigQuery then perform distributed rule-based reasoning to infer all the

explicit knowledge in the dataset. This rule-based reasoning is achieved by using a combination of processors and BigQuery, which will be explained in Section 6.4. Ultimately, once the reasoning process is completed, the dataset is ready to be queried using BigQuery queries, or SPARQL queries if implemented. However, before the dataset is loaded into BigQuery, it needs to be transformed into a format that BigQuery understands, such as JSON or CSV. Additionally, as BigQuery charges for queries by the size and type of the scanned data, the cost of processing and storage is reduced by encoding the data from string to integer values using dictionary encoding. Moreover, to distribute the rule-based reasoning between a number processors the dataset needs to be analysed upfront — as mentioned in Section 6.1.1 — to devise a partitioning strategy for forward rule-based reasoning.

6.2 Dictionary Encoding

This section forms one of the contributions of the thesis and provides answers to the research question "Can an efficient dictionary that fits in-memory be created for encoding RDF URIs from strings to integers?". The focus in this section is on creating an efficient dictionary that can be held in memory, such a dictionary can speedup the RDF dataset encoding and loading processes considerably, compared to other approaches that generate large in-file dictionaries. Additionally, the ability to hold the entire dictionary in memory makes it easy to update the dictionary after creation, furthermore, applications can readily encode and decode the dataset queries. Most importantly, dictionary encoding reduces the dataset size, subsequently reducing the cost of the dataset storage and processing.

As noted in Section 3.4.3.3, that BigQuery charges for queries by the size of the columns scanned. For example, the size of each cell in a column of type string will be

2 bytes + the UTF-8 encoded string size, the scanned data size can amount to many gigabytes and terabytes when dealing with massive datasets with long URIRefs. This cost can rapidly be multiplied when issuing a large number of queries as part of the distributed reasoning process that will be described in Section 6.3.

6.2.1 Reducing the Size of URIRefs

BigQuery supports 64-bit signed integers, with the most significant bit reserved as a sign bit, the minimum number that can be stored is -2^{63} and the maximum is $2^{63} - 1$. To highlight the potential savings when converting URIs from strings to integers, consider the following URIRefs from the running example presented in Figure 2.1:

1. `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`
2. `http://inetria.org/directory/schema#Employee`
3. `http://inetria.org/directory/schema#Company`
4. `http://inetria.org/directory/employee/smithj`
5. `http://inetria.org/directory/employee/doej`

The aforementioned URIRefs contain on average 44 characters per URIRef, with UTF 8 encoding each character can consume from 1 to 4 bytes for storage. On average, this equates to a minimum of 46 bytes for each URIRef, with integer encoding, only 8 bytes are required to store each value, equating to a storage saving of at least 82%. With this evident savings on both storage and cost, a straightforward dictionary encoding approach similar to [67] can be employed. However, one of the major issues with such approach due to redundancy, is the size of the dictionary, which prohibit in-memory handling of the dictionary and instead resorts to in-file processing, resulting in a considerable processing overhead.

This section proposes an efficient approach to reduce the size of the dictionary and therefore be able to load it into memory to speedup the encoding process. It is worth

Table 6.1: Encoded example of the first two rows in Table 6.3

Subject	**Predicate**	**Object**	**Object_Literal**
345	50	250	NULL
345	300	NULL	”John Smith”

noting that this approach does not compress literal values, according to Formula 2.1 a literal can not occur on the subject or predicate term and non of the BigQuery reasoning queries apart from rule *rdfs7* will retrieve literals, as will be explained in Section 6.3.3. Additionally, by not encoding literals, the BigQuery string functions can be used to search for particular text, this is not the case with URIRefs as they are merely identifiers which are queried in their entirety. To store the encoded data, a BigQuery *triple table* is used, an example is shown in Table 6.1. This table has three columns *predicate, subject and object* of type integer, these will hold the encoded values of the original URIRefs and blank nodes. A fourth column *object_literal* of type string is added to store any literals as they are, in which case the *object* column will contain a *NULL* value.

6.2.2 Efficient URI Reference Compression

This section introduces an approach for compressing URIRef in NTriple documents using integer values to build up an efficient (compact) dictionary. Consider for example the URIRefs numbers 4 and 5 in the previous list, it can be seen that they both share the `http://inetria.org/directory/employee` URI. In a large dataset, many more terms will share similar URIs, which, in a straightforward dictionary encoding the dictionary will contain the URI many times, thus inflating the dictionary with redundancy. If each of the URIRefs are split into two parts, the first contains the hostname and part of the path (e.g. `inetria.org/directory/employee`) and the second contains the last part of the path (e.g. `smithj` and `doej`). The dictionary will

only contain the following parts: `inetria.org/directory/employee`, `smithj` and `doej`, instead of the full URIRefs in 4 and 5 in the previous list.

It can be seen that this approach can reduce the size of the dictionary, noting that the scheme (e.g. http(s)://) and the NTriple angle brackets "⟨" and "⟩" have been removed as these can be added later when decoding the data. URIRefs that belong to the RDF, RDFS and OWL vocabulary such as item 1 in the previous list (`http://www.w3.org/1999/02/22-rdf-syntax-ns#type`) are limited and are hence encoded in their entirety — using a set of fixed integer values — without being split.

6.2.3 Encoding Terms

To reduce redundancy in the dictionary, the URIRefs can be split on a particular location, this thesis uses the last path separator "/" as the standard approach, however, other strategies are evaluated in Chapter 7 as well. These can be summarised as follows:

- **Standard**, the URIRefs are split on the last path separator "/", this is the default strategy.
- **Hostname**, the URIRefs are split on the first path separator "/" after the hostname.
- **Hostname + first path part (1stPP)**, the URIRefs are split on the second path separator "/" after the hostname.

Using these split strategies means most encoded URIRefs will now have two parts, both are encoded as integers. A question that begs asking, is how can these two integer values stored together in one value that can be stored in a single BigQuery column?, a novel approach is to use a "bitwise" pairing function [131]. Such a function

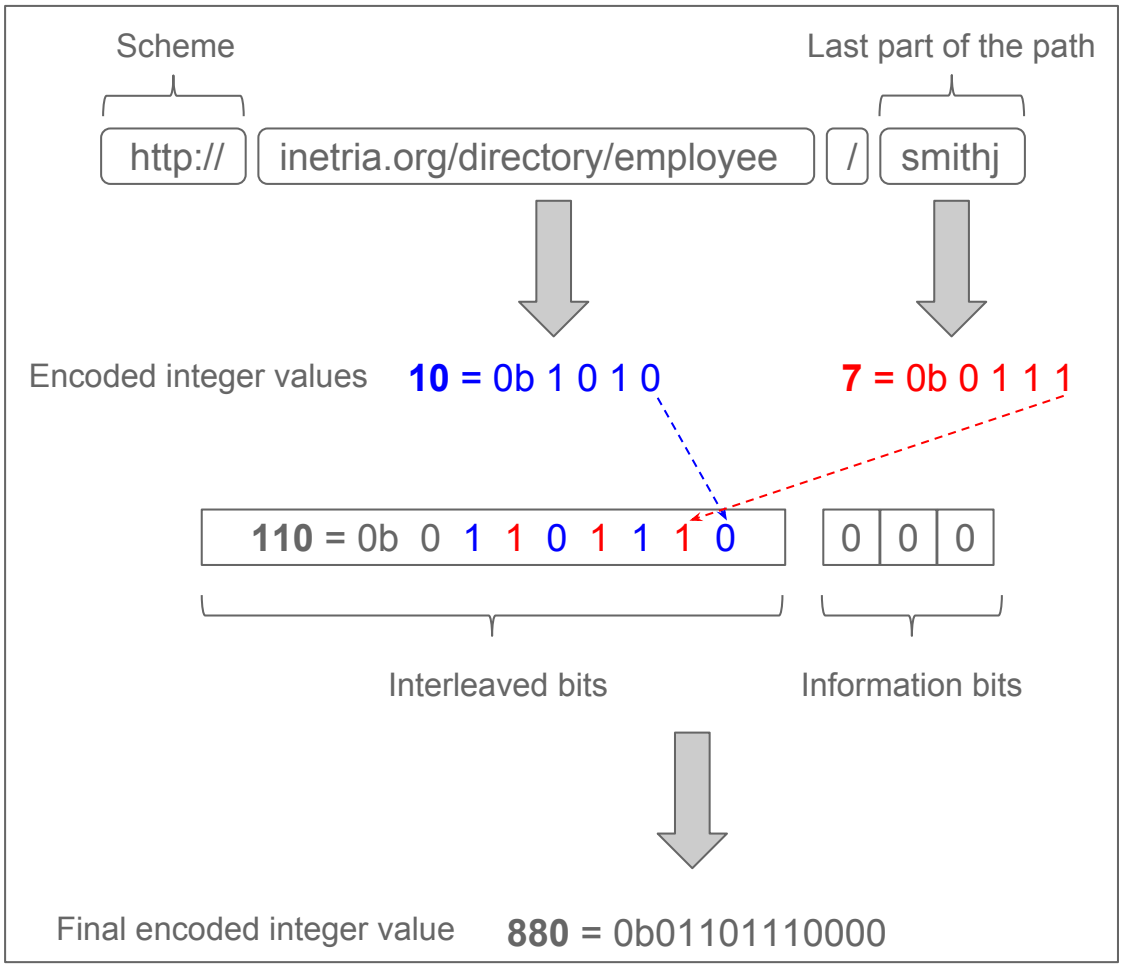


Figure 6.3: Dictionary Encoding Using Binary Interleaving

Table 6.2: Information Bits for URI Reference Compression

First bit - parts bit (least significant bit)	0	URIRef has two parts (e.g. http://inetria.org/directory)
	1	URIRef has one part (e.g. http://inetria.org or blank node _:jA5492297)
Second bit - scheme bit	0	Scheme is http
	1	Scheme is https
Third bit - slash bit	0	URIRef does not have slash (/) at the end (e.g. http://inetria.org)
	1	URIRef has a slash at the end (e.g. http://inetria.org/)

is based on bit interleaving of two number so that they can be joined into one with the reverse process extracting back the two numbers. Figure 6.3 shows an example of encoding the URIRef `http://inetria.org/directory/employee/smithj` using the dictionary encoding algorithm outlined here. As seen from the example, firstly the scheme is removed, then the remaining part of the URIRef is split at the last path separator. This results in two strings that are encoded using incremental integer values (e.g. 10 and 7). These two values are then paired by interleaving their bits resulting in the decimal number 110. Finally, three least significant information bits (summarised in Table 6.2) are added to store information about the original URIRef, resulting in a final encoded decimal of 880.

6.2.4 Decoding Terms

Pairing is only used when a term is a URIRef with more than one section. Terms that do not contain multiple parts such as blank nodes are encoded into integer values with the additional information bits added. When decoding integer values, firstly the three least significant information bits are extracted, if the parts bit is 1 (Table 6.2), then the integer value after removing these three bits can immediately be resolved from the dictionary. If the parts bit is 0 then the integer values of the two parts are unpaired by restoring their bits, the resultant values are then retrieved from the dictionary and concatenated. The scheme bit determines if *http* or *https* should be added to the decoded string, similarly, the slash bit is used to add a slash "/" to the end of the URIRef. Finally the NTriple angle brackets "⟨" and "⟩" are added to the URIRef to complete the term decoding.

6.2.5 Storage Considerations

As noted earlier, BigQuery supports 64-bit signed integers, the minimum number that can be stored is -2^{63} and the maximum is $2^{63} - 1$. Moreover, three additional bits are reserved for storing information regarding the encoded URIRef, this leaves the user with a minimum and maximum between -2^{60} and $2^{60} - 1$. For the dictionary encoding approach, bit interleaving might be needed for encoding URIRefs that were split into two parts. Bit interleaving will result in a number that contains the sum of the bits in the two joined numbers, this mean, in order to ensure that the bit interleaved numbers do not exceed the maximum of $2^{60} - 1$. Each of the joined parts must be between -2^{30} and $2^{30} - 1$, giving users the ability to encode string parts using integer values that ranges from $-1,073,741,824$ to $1,073,741,823$. With the dictionary using both positive and negative integers within this range, this approach can encode approximately 2.14 billion unique URI parts.

6.2.6 Dictionary Encoding Tasks

To perform the dictionary encoding algorithms outlined in the previous sections, the following CloudEx tasks have been created and explained in the next sections:

- Extract Terms Task
- Assemble Dictionary Task
- Encode Data Task

6.2.6.1 Extract Terms Task

This task is distributed between a number of CloudEx processors to process the dataset files and extract the URI parts from each file. This process will extract the URIs and split them by using one of the split strategies outlined in Section 6.2.3. The

extracted parts for each of the dataset files are compressed and saved in a separate file and uploaded to Cloud Storage.

6.2.6.2 AssembleDictionaryTask

This task is run on a single processor with high memory configurations if required to assemble the dictionary in memory. The files extracted as part of the previous step are loaded and the terms are added to an in-memory dictionary hash table, assigning each term a unique integer identifier. The identifiers are derived from an incremental counter. This assembly process can be parallelized by using multiple threads to add the parts from multiple files to the dictionary concurrently. Once the dictionary assembly is completed, the dictionary is compressed, serialised to disk and uploaded to Cloud Storage.

6.2.6.3 Encode Data Task

Similar to the first task, this task is distributed between a number CloudEx processors to encode the dataset files. Each processor downloads the dictionary from Cloud Storage and loads it in memory. For each file the URIs are extracted, split using one of the split strategies, encoded and written to a compressed output file. The output file is written in CSV format such that it can be imported to Google BigQuery. These output files are then uploaded by each processor to Cloud Storage.

6.3 Distributed RDFS Reasoning

This section forms one of the contributions of the thesis and provides answers to the research question "Can cloud-based big data services such as Google BigQuery be used to perform RDFS rule-based reasoning?". This section is focused on utilising big data

columnar databases in general and Google BigQuery in particular to perform RDFS rule-based reasoning. The use of big data databases to perform rule-based reasoning utilises their analytical capabilities to rapidly query billions of statements. These capabilities shift some of the processing from computing nodes over to the databases, hence eliminating many challenges with workload distribution. Additionally, as the original and inferred data is available in such a database it can be queried immediately. This is an improvement over approaches that output the data in files that are yet to be imported into a system that supports querying. The ability to perform forward reasoning on large datasets with a billion or more statements on the cloud, enables applications large and small to utilise RDF capabilities.

The primary objective of processing RDF datasets using ECARF is to load such datasets into BigQuery, hence enabling users to query them. BigQuery utilises many servers internally to parallelize query execution and results aggregation, a process that is obscure to the end user. Unlike a traditional relational database, there are no user generated indexes or any means of tuning this query execution. For this reason, BigQuery is treated as a black box that can execute queries over massive datasets in parallel and return their results in reasonable time (less than 10 seconds). This section describes the approach used for performing forward RDFS reasoning by utilising BigQuery and a number of processing nodes. Firstly this section explains how rule based reasoning can be performed using BigQuery, then it outlines the strategy used for distributing this reasoning between a number of processors.

6.3.1 Handling Schema Triples

As mentioned in Chapter 2, the schema data of the RDF datasets on the web represents only a small fraction of the overall data [70] and can easily fit in memory [85]. Therefore, the closure of the schema data is computed upfront either offline or using

Table 6.3: Example Triple Table for the Instance Data in Figure 2.1(c)

Subject	**Predicate**	**Object**
employee:smithj	rdf:type	inetria:Employee
employee:smithj	foaf:name	”John Smith”
employee:smithj	inetria:manages	employee:doej
employee:smithj	inetria:works_at	business:inetria
employee:smithj	inetria:start_date	”2002-04-17”^^xsd:date
employee:smithj	inetria:drives	_:jA5492297
_:jA5492297	inetria:make	”Ford”
_:jA5492297	inetria:reg_number	”ABC123”
employee:doej	rdf:type	inetria:Employee
employee:doej	inetria:works_at	business:inetria
employee:doej	foaf:name	”John Doe”

a coordinator task that applies the schema rules in Table 2.2. Consider for example the schema triples shown in Figure 2.1(c), due to the small size of this data there is no need to distribute its processing. Rules *rdfs5* and *rdfs11* can simply be applied either offline by using off-the-shelf reasoner like Apache Jena[2], or carried out using a coordinator task.

6.3.2 Handling Instance Triples

In terms of instance triples they are loaded into a BigQuery table that called a *triple table*, which has three columns *subject*, *predicate* and *object*, or with a fourth *object_literal* column if dictionary encoding is used (Section 6.2.2). An example of such table is shown in Table 6.3 for the instance data presented in Figure 2.1(d). This table can be queried using SQL-like queries in the following format:

```
SELECT [subject] [,predicate] [,object]
FROM triple_table WHERE [condition];
```

[2] `https://jena.apache.org/`

6.3.3 Performing Forward Reasoning Using BigQuery

Once all the instance triples are loaded into the *triple table*, the forward rule-based reasoning can be started by using the *SI* rules summarised in Table 2.1, namely *rdfs2*, *rdfs3*, *rdfs7* and *rdfs9*, all of which have two triples in the body, a schema triple and an instance triple. Since all the schema triples are available in memory and the instance triple are loaded in the *triple table*, for each rule the reasoning can be started from the schema triples. This approach can iterate through the schema triples and for each triple query the *triple table* for the instance triples terms that are needed to generate the triple in the head of the rule.

This reasoning process can be explained by means of a worked example using the dataset in Figure 2.1. Consider the following two schema triples, both share the subject term `inetria:works_at`:

```
inetria:works_at rdfs:domain inetria:Employee
inetria:works_at rdfs:range inetria:Company
```

For the first schema triple rule *rdfs2* and for the second rule *rdfs3* can be applied respectively. If the second schema triple and rule *rdfs3* are considered, to generate the head of this rule in the form `?y rdf:type ?c`, the *triple table* needs to be queried for any instance triples in the form `?x inetria:works_at ?y`. Note, the variable `?p` in the second part of the rule's body is replaced with `inetria:works_at`. Since the head triple does not contain `?x`, it is only needed to query for `?y` to generate a triple in the form `?y rdfs:type inetria:Company`, with the variable `?c` replaced with `inetria:Company`. To query for the instance triples needed to generate the head for rule *rdfs3*, the following query can be issued:

```
SELECT object FROM triple_table
WHERE predicate = 'inetria:works_at';
```

Table 6.4: BigQuery SQL-like queries for the *SI* rules reasoning, (*sc = subClassOf, sp = subPropertyOf, dom = domain*)

Rule	Body		Select clause Columns	Where clause Condition
	Schema Triple	Instance Triple		
rdfs2	?p, rdfs:dom, ?c	?x, ?p, ?y	subject	predicate = ?p
rdfs3	?p, rdfs:range, ?c	?x, ?p, ?y	object	predicate = ?p
rdfs7	?p1, rdfs:sp, ?p2	?x, ?p1, ?y	subject, object	predicate = ?p1
rdfs9	?c1, rdfs:sc, ?c2	?x, rdf:type, ?c1	subject	predicate = rdf:type and object = ?c1

This query returns `business:inetria` as the result, which can be verified from Table 6.3 that two of the triples have `inetria:works_at` as a predicate. Note that this query will return two similar results which when used will result in two duplicate triples being generated. This duplicates issue has been extensively discussed in the literature [18, 19, 20, 21], with some approaches eliminating duplicates before or after being generated. Duplicates results in redundant data and hence extra storage space. Chapter 7 evaluates a memory based approach to eliminating duplicates for small datasets.

To return to the subject, from the returned query result the head of the rule can be generated as `business:inetria rdfs:type inetria:Company`. This inferred triple (new knowledge) is then inserted back into the triple table as explained in Section 6.4.8. Now, if the first schema triple mentioned previously and rule *rdfs2* are considered, it is clear in this case the query should select `?x` rather than `?y`. Since the two schema triples share the same subject, the reasoning process for these two triples can be carried out together by combining their queries into one to avoid making unnecessary queries, which in turn will reduce cost:

```
SELECT subject, object FROM triple_table
WHERE predicate = 'inetria:works_at';
```

6.3.3.1 The Relevant Term

If the other *SI* rules in Table 2.1 are inspected, it is clear that the same reasoning approach presented here can be followed for the remaining rules. It is also apparent that the subject term of the schema triples can be used in the `WHERE` clause of the query as highlighted in Yellow in Table 6.4. This term is called the *relevant* term because it drives the reasoning process and thereby if no instance triples are found when querying by this term, no reasoning will take place. Consequently, if the instance data is analysed upfront it can readily be determined which schema relevant terms will drive inference and which will not. Additionally, the occurrences of instance triples for each of the relevant terms can be counted, with this count then utilised to partition the reasoning process between a number of processors. The following definitions can be provided based on the approach described here which uses the relevant schema terms to drive the reasoning process:

Definition 6.3.1. ***Relevant schema term****, is the term in a schema triple that can be used for the reasoning and partitioning strategy. For the rules considered in this thesis (Table 2.1), this term is always the subject term.*

Definition 6.3.2. ***Productive schema term****, is a relevant term that will drive inference. This means there are instance triples that contain this term and their data will be returned when issuing the reasoning queries described in Section 6.3.3.*

Definition 6.3.3. ***Productive instance triple****, is a triple that will take part in the reasoning process for one of the productive schema terms. Data for such triple will be returned as part of the reasoning queries run against the triple table.*

Definition 6.3.4. ***Selective term partitioning****, is the name of the approach outlined here for partitioning the reasoning workload between a number of processors by*

using relevant schema terms, upfront analysis of the instance data and bin packing algorithm.

6.3.3.2 Query Optimisation

Some of the *SI* rules in Table 2.1 require different data to generate the head, so the `SELECT` clause may contain different columns as shown in Red in Table 6.4. Also, as noted earlier, if all the schema triples that contain the relevant term as subject are grouped together, the reasoning for them can be performed as a group by issuing a single query to retrieve the data needed to generate the heads of the rules. For example, if a term has both `rdfs:domain` and `rdfs:range` schema triples, one query can be issued with a select clause of `SELECT subject, object` rather than two separate queries to reason for both rules *rdfs2* and *rdfs3*.

As noted in Section 3.4.3.3, the cost of using Google BigQuery is dependent on the size of the scanned data, which in this case is the size of one or two columns for each of the productive terms. For bigger datasets with many productive terms this approach can be very costly and results in BigQuery scanning terabytes of data. Since the maximum size of a BigQuery query is 256 KB (Section 3.4.3.1), an approach can be utilised such that each processor joins all the queries for its productive schema terms into one query, which will reduce the size of scanned data and consequently the cost. However, with a single query it is required to select all the three columns subject, predicate and object to be able to conduct the rule-based reasoning. This results in more data to transfer from BigQuery to the processors, but with dictionary encoding the size of this data is reduced considerably.

6.3.3.3 Distributing the Reasoning Process

Based on the approach described in the previous section, the reasoning process can be partitioned between multiple processors by assigning each one of them a number of productive schema terms. Since the count of instance triples for each relevant term is already known — by analysing the instance data upfront — this count can be used as the basis for the workload partitioning strategy. The reasoning partitioning strategy is based on assigning each processor a number of schema terms that amount to an equal share of instance triples. Each processor then loads in memory the schema triples related to the schema terms assigned to it and as described in this section, uses the schema triples to query the *triple table* accordingly.

As it was noted in Section 6.1.3, the instance data needs to be pre-processed before hand to convert it to CSV, a format that BigQuery supports. This pre-processing is utilised to count the instance triples for each of the relevant terms, but before doing so, the schema data needs to be analysed to extract the relevant terms. Both of these approaches are explained in the following sections.

6.3.4 The Schema Terms Analysis Step

The first step towards partitioning the reasoning process is to analyse the schema triples to extract the relevant terms. This only analyses the *subject* terms of the schema triples that match the schema triple format of the *SI* rules' bodies. More specifically, any schema triples that have the following as predicate: `rdfs:domain`, `rdfs:range`, `rdfs:subPropertyOf` and `rdfs:subClassOf`. As noted in the previous section, the *subject* term of a schema triple is the minimum that is needed to query the relevant instance triples data in the *triple table*. The schema triples are analysed by a CloudEx coordinator task and grouped by their subject as illustrated in Algorithm

Algorithm 3 Analyse the schema terms

```
1: procedure AnalyseSchemaTerms(T)                    ▷ T is a set of all schema triples
2:     R ← {}                                         ▷ The dictionary(map) of term:triples pairs
3:     for all triple ∈ T do
4:         predicate ← triple.predicate
5:         subject ← triple.subject
6:         if predicate ∈ {"dom", "range", "sc", "sp"} then
7:             L ← {triple}
8:             if subject ∈ R.keys then
9:                 L ← R.get(subject) ∪ L
10:            end if
11:            R.put(subject, L)
12:        end if
13:    end for
14:    return R
15: end procedure
```

3. This grouping is done so that each processor can be given a list of schema terms alongside their schema triples to be used for counting the relevant instance triples.

6.3.5 The Instance Triples Count Step

Once the relevant schema terms are extracted, the count step can be distributed between a number of processors. Each processor uses the schema terms extracted by the coordinator to analyse all the instance triples and count their occurrences for each of the schema terms as illustrated in Algorithm 4. To be more specific, each processor will only count the instance triples that contain the relevant term as a predicate or object as highlighted in Yellow on the "Instance Triple" column in Table 6.4. At the same time, the processors can extract the URI parts for the dictionary as was explained in Section 6.2.

Once all the processors have finished analysing the data, the results from each are merged by the coordinator to get an overall count of the instance triples for each of the schema terms. This count is used later for partitioning the reasoning process as will be discussed in Section 6.3.6. Any schema terms that do not have any instance

Algorithm 4 Count the relevant instance triples

```
1: procedure CountInstanceTriples(T, S) ▷ T is a set of allocated instance triples ▷ S is a set relevant schema terms
2:     C ← {}                                   ▷ The dictionary(map) of term:count pairs
3:     for all triple ∈ T do
4:         for all term ∈ {triple.subject, triple.predicate, triple.object} do
5:             if term ∈ S then
6:                 i ← 1
7:                 if term ∈ C.keys then
8:                     i ← C.get(term) + i
9:                 end if
10:                C.put(term, i)
11:            end if
12:        end for
13:    end for
14:    return C
15: end procedure
```

triple count are not considered as they will not drive any inference.

6.3.6 Workload Partitioning

ECARF uses the CloudEx framework Bin Packing partitioning function explained in Section 5.3. In order to use this partitioning approach the size of each workload item needs to be quantified. For the approaches described in this chapter two cases are considered, firstly when distributing the processing of the RDF dataset files and secondly when distributing the reasoning process. For the first case either the size of each file (compressed or uncompressed), or the number of statements in each file can be used, however experimental evaluation is conducted in Chapter 7 to determine this partitioning factor for each of the processing tasks. For the second case, when distributing the reasoning process, the count of productive instance triples is used, similarly the effectiveness of this strategy is evaluated in Chapter 7.

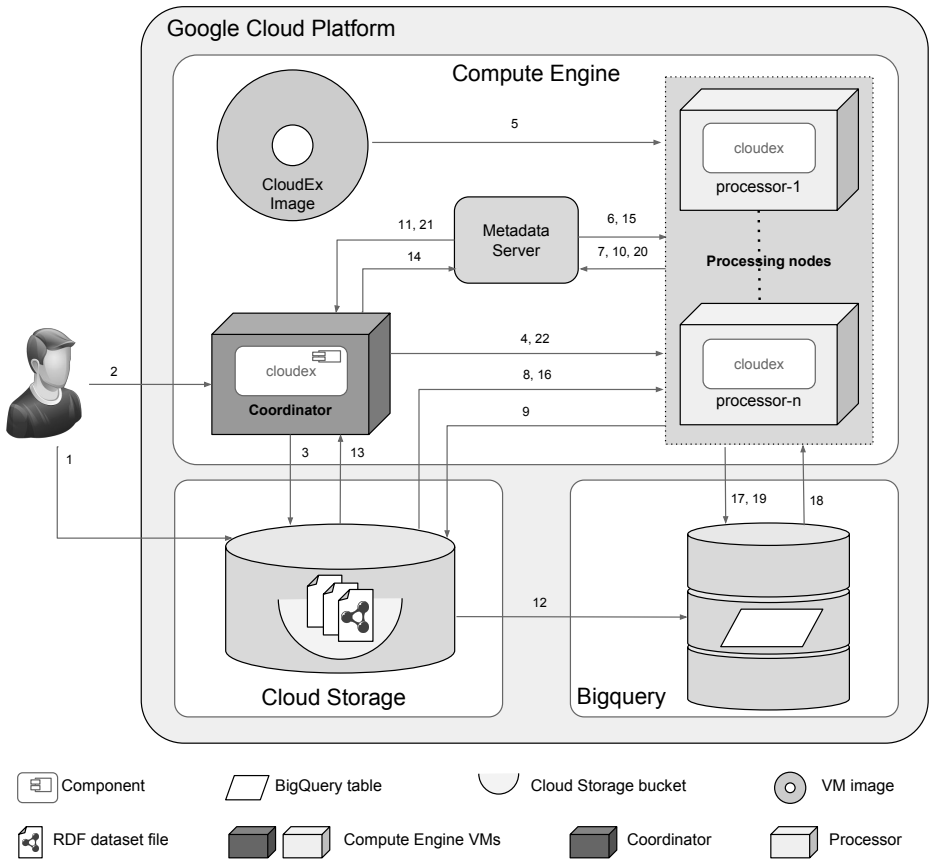


Figure 6.4: ECARF Architecture Walkthrough

6.4 ECARF Architecture Walkthrough

This section provides a walkthrough the ECARF architecture shown in Figure 6.4. To better illustrate the overall architecture and interactions, numbered arrows are used to indicate the sequence of steps and are referenced throughout the following sections using round brackets (e.g. *(1)*). To load an RDF dataset into BigQuery and perform forward RDFS reasoning, the user uploads the Gzip-compressed dataset files into Cloud Storage (1). These files contain the instance triples of the dataset, the schema triples are uploaded in a separate file (Section 6.3.1).

If the coordinator is not already running the user starts it using the Google Cloud

Console[3]. Once started the coordinator is supplied with the name of the Cloud Storage bucket containing the dataset, the name of the schema triples file and the BigQuery table to use (2). The following sections explain in details the various tasks performed by ECARF, which are also illustrated in Figures 6.5, 6.6 and 6.7 as CloudEx tasks using the notation developed in Chapter 5.

6.4.1 The Schema Term Analysis Task

Initially the coordinator analyses the schema triples file and extracts all the relevant schema terms as explained in Section 6.3.4 and illustrated in Algorithm 3. As noted previously, for the RDF data on the Web the schema data is very small compared to the overall data size ($\simeq 2\%$), hence the coordinator can readily process this data in memory. The coordinator generates a map using the relevant terms as keys and a list of the triples containing these terms as values. The coordinator then serialises this map to a file and uploads it to Cloud Storage as illustrated in Figure 6.5.

6.4.2 Distributing the Count Task

After processing the schema, the coordinator lists the dataset files in the bucket (3) and uses the task partitioning function to calculate the number of processors to use. The weight of each file is set as the compressed size of the file. Based on the number of bins N calculated by the task partitioning function ($f1$ in Figure 6.5), the coordinator starts N processors by issuing *Instances.insert* Compute Engine API call (4). This API call includes a reference to the previously created processor VM image I and the initial metadata for each of the processors to perform the "instance triple count / extract dictionary parts" task. The provided metadata includes a list of files to process, the names of the schema terms file, the bucket containing the dataset and a

[3] `https://console.developers.google.com/`

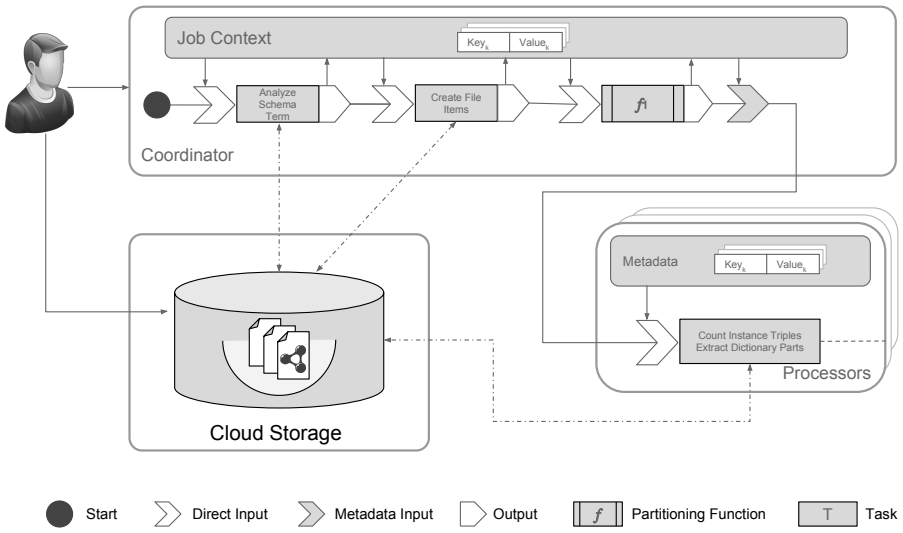


Figure 6.5: ECARF Schema Terms Analysis and Distribute Count Tasks

startup script, an example is shown in Table 6.5.

Table 6.5: Sample ECARF *instance triples count / extract dictionary parts* task processor metadata

Key	**Value**
ecarf-task	ExtractCountTerms2PartTask
ecarf-bucket	dataset
ecarf-files	dataset_file1.nt.gz,dataset_file2.nt.gz,...
ecarf-schema-terms	schema_terms.json
startup-script	/home/ecarf/start_ecarf.sh

6.4.3 The Instance Triple Count / Extract Dictionary Parts Task

Compute Engine uses the predefined VM image I to create the processors (5), once started each processor executes the startup script which starts the *cloudex* application. A startup script is used to run the *cloudex* instead of hard-coding it in the VM image I, which provides the flexibility of downloading an updated application or performing

arbitrary operations when the processor starts up. When the *cloudex* application runs it retrieves the instance metadata (6) then updates it to indicate that the processor is busy (e.g. `ecarf-status=BUSY`) (7).

The "instance triple count and extract dictionary parts" task — illustrated in Figure 6.5 — downloads the schema terms file and the assigned instance triples files from Cloud Storage (8) then loads the relevant schema terms in memory. It then processes the instance triples files and extracts the dictionary URI parts as outlined in Section 6.2.3, this is done whilst counting the instance triples for each of the relevant schema terms as illustrated in Algorithm 4. The extracted URI parts for each of the dataset are compressed in memory and written to output files, once done, the task uploads the output files with a file containing the summary of the total number of triples for each of the schema terms to Cloud Storage (9). The *cloudex* application then updates the metadata to indicate that the processor is free (e.g. `ecarf-status=READY`) (10) and awaits for metadata changes.

6.4.4 Assemble Dictionary Task

As shown in Figure 6.6, the coordinator uses the task partitioning function for this task ($f2$), which is hardcoded to use a single processor. Once the task is delegated to the processor through the metadata server, it downloads the URI parts files generated previously in Cloud Storage, then merges these parts into a dictionary as illustrated in Figure 6.6. The keys for the dictionary are generated incrementally, but as noted previously the keys for any of the RDF, RDFS and OWL vocabulary URIRefs are set to prefixed values. Once the processor completes the dictionary generation process, the dictionary is compressed in-memory, serialised to disk and then uploaded to Cloud Storage. The processor then updates its metadata to indicate its availability, subsequently the coordinator starts the next task discussed in the following section.

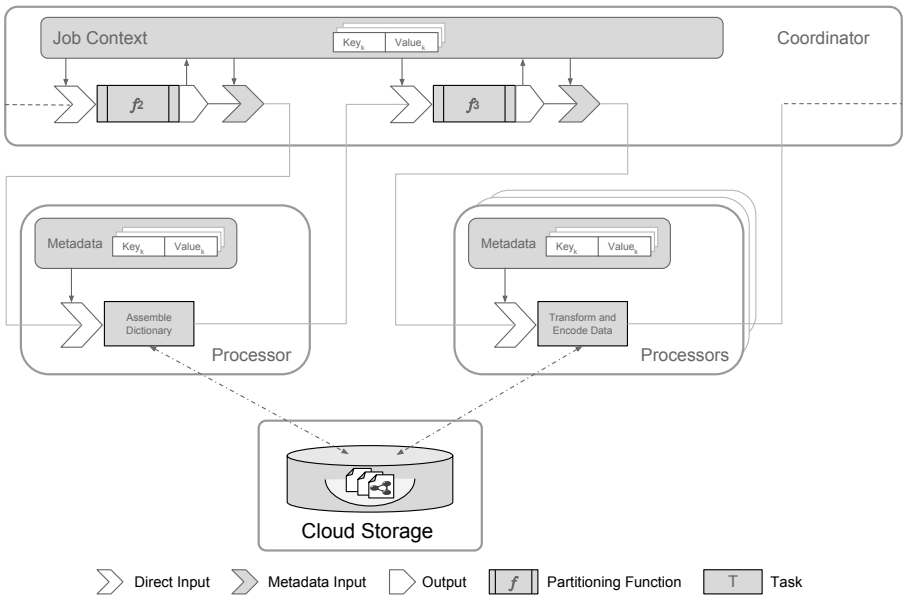


Figure 6.6: ECARF Assemble Dictionary and Transform/Encode Data Tasks

6.4.5 Transform and Encode Data Tasks

The coordinator uses the task's partitioning function ($f3$) to distribute its processing. If the number of processors required exceeds the number of processors already running, new processors are started to account for the difference. In order to encode the data, the processors download the dictionary from Cloud Storage and load the entire dictionary in main memory. The processors then process the dataset files, encoding each of the terms and formatting the encoded terms as CSV before saving the results to Gzip compressed files as illustrated in Figure 6.6. These files are uploaded to Cloud Storage and when the process is completed each processor updates its metadata to indicate its availability. The coordinator then starts the next task.

6.4.6 Aggregate Processors Results Task

The coordinator regularly checks the metadata of each processor by issuing *Instances.get* Compute Engine API call (11). Once all the processors have completed processing

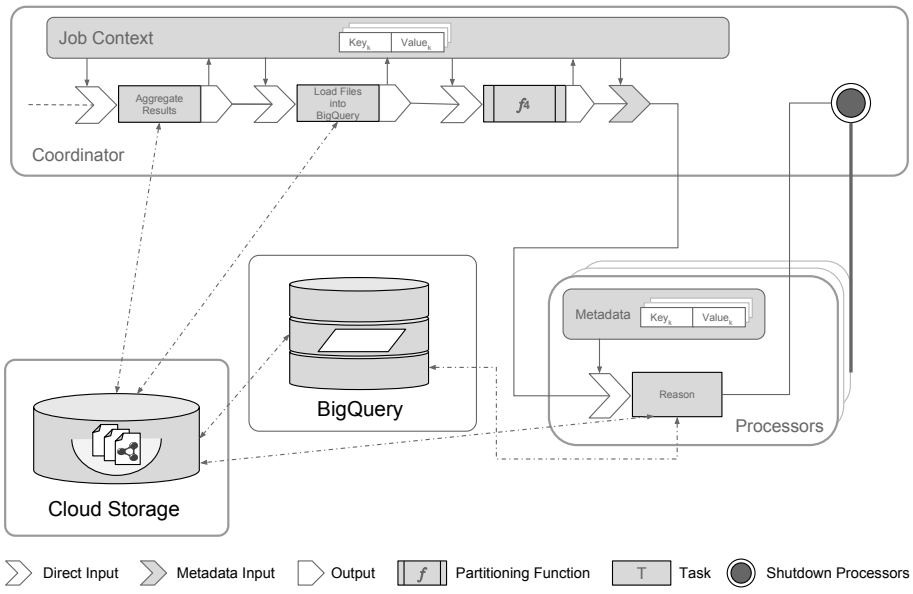


Figure 6.7: ECARF Aggregate Results and Reason Tasks

and analysing the instance data, the coordinator downloads the summary files and merges their results (13). These summary files contain the count of instance triples for each of the relevant schema terms. Based on the total count for each term, the coordinator uses the reason task partitioning function ($f4$ in Figure 6.7) to calculate the number of processors to use for the distributed reasoning process. If the number of processors required exceeds the number of processors already running, new processors are started to account for the difference.

6.4.7 Load Files into BigQuery Task

For this task — illustrated in Figure 6.7 — the coordinator invokes *Jobs.insert* BigQuery API to load into BigQuery the CSV dataset files generated by the processors during the transform and encode data task. This API call provides the table name, table schema and a list of the processed dataset files available in Cloud Storage. Subsequently, BigQuery uploads all the processed CSV files into a new table (12). The

coordinator then updates the metadata of each processor with the name of the BigQuery table, the schema data file and the productive schema terms that the processor should use for reasoning (14). The metadata server then notifies each processor of these metadata changes (15), this is achieved by using a hanging HTTP GET as described in Section 3.4.1.1.

6.4.8 The Forward Reasoning Task

When this task runs, it downloads the schema triples file from Cloud Storage (16) by issuing a *Objects.get* Cloud Storage API call. Each processor then performs rule-based reasoning by issuing a number of BigQuery queries using the *Jobs.insert* BigQuery API operation (17). This process was explained in detail in Section 6.3.3 and illustrated in Algorithm 5. The queries run asynchronously and once done the processor retrieves the results of the query if any (18). The processor then applies the rules in Table 6.4 to infer new triples and uploads them to the the BigQuery table (19).
The processors then sleeps for a number of seconds before issuing the same queries again using BigQuery *table decorators* (Section 3.4.3.1) to retrieve any of the inferred triples that were inserted by other processors. If no new data is retrieved from the BigQuery table after a number of retries then the reasoning process is complete and the processor updates its metadata to indicate its availability (20). Once the coordinator detects from the processors metadata (21) that they are available, it shuts them down — by issuing a Compute Engine *Instances.delete* operation — to avoid incurring unnecessary cost (22) as illustrated in Figure 6.7. This concludes the reasoning task, both the original and inferred triples are now in a BigQuery table and can be queried by executing BigQuery queries.

Algorithm 5 Triple table reasoning over instance triples

```
1: procedure InferInstanceTriples(D) ▷ D - dictionary of schema term:triples pairs ▷ Build query dynamically
2:     query ← "SELECT subject, predicate, object FROM triple_table WHERE "
3:     predicate ← "predicate IN "
4:     object ← "object IN "
5:     for all term ∈ D.keys do
6:         triples ← D.get(term)
7:         for all triple ∈ triples do
8:             if triple.predicate = "sc" then
9:                 object ← object + triple.subject
10:            else
11:                predicate ← predicate + triple.subject
12:            end if
13:        end for
14:    end for
                                                    ▷ Retrieve results from table
15:    db_triples ← queryTripleTable(query + predicate + object)
                                                    ▷ Infer new triples
16:    for all term ∈ D.keys do
17:        triples ← D.get(term)
18:        for all triple ∈ triples do
19:            for all db_triple ∈ db_triples do
20:                Write fireSIRule(triple, db_triple)
21:            end for
22:        end for
23:    end for
24: end procedure
```

6.5 Summary

This chapter has addressed the issues outlined in Chapter 2 with distributed RDF processing, namely dictionary encoding, data storage and workload partitioning. The dictionary encoding algorithm addresses the issues with existing approaches by providing an efficient dictionary that can fit in memory to speed up the encoding process. Additionally, the data storage issue is addressed by moving the storage from the processors over to cloud services such as storage and big data. Furthermore, the workload partitioning issue is addressed by using the CloudEx's Bin Packing function and the selective term partitioning strategy outlined in this chapter. This chapter has fo-

cused on the remaining contributions of this thesis to answer the follow two research questions:

- Can an efficient dictionary that fits in-memory be created for encoding RDF URIs from strings to integers?
- Can cloud-based big data services such as Google BigQuery be used to perform RDFS rule-based reasoning?

The chapter presented ECARF, a generic and extendible cloud based triple store for RDFS processing built on top of the CloudEx framework. The first research question was answered by presenting an algorithm for creating efficient dictionaries for RDF datasets that can fit in memory. With the growing size of RDF dataset to billions of statements, dictionary encoding is crucial to reduce the overall dataset size, and to speed up its processing. Since this research is focused on using cloud computing, the encoding also helps to compress the data and subsequently reduce the cost of storage and processing.

The second research question was answered by implementing an algorithm that performs forward RDFS reasoning using big data columnar databases in general and Google BigQuery in particular. This algorithm utilises the analytical capabilities of such databases to offload some of the processing from computing nodes, hence benefiting form the extra processing power. This algorithm, alongside the aforementioned dictionary algorithm and CloudEx features, provides a triple store solution that applications can use without the utilisation of high specification hardware. Moreover, by using BigQuery for storing the processed RDF datasets, the data can immediately be queried, which is an added advantage over solutions that output files as a result of the forward reasoning process. The next chapter conducts experimental evaluation of the algorithms presented in this chapter to measure their effectiveness.

Chapter 7

Evaluation of Cloud Based RDF Processing

The aim of this research is to develop and evaluate an elastic cloud-based RDF triple store to process large RDF datasets. In order to efficiently utilise cloud computing features and achieve this aim, Chapter 5 presented CloudEx, a generic cloud-first task execution framework based on a divide-conquer approach. Subsequently, Chapter 6 introduced the ECARF triple store, which is a set of algorithms implemented as CloudEx tasks and jobs to efficiently process RDF datasets on the cloud. Both of these chapters have presented novel algorithms that form the contributions of this thesis and answer the research questions.

This chapter evaluates these algorithms and the CloudEx framework within the scope of distributed processing of RDF datasets on the Google Cloud Platform. This chapter reports on the experiments conducted to evaluate the algorithms developed in Chapters 5 and 6, the chapter also analyses and discusses the results and findings. Additionally, where possible, this chapter provides comparison with other related approaches. Furthermore, as part of the evaluation phase, a number of optimisations

have been made in order to reduce the cost and time of executing the algorithms developed in this thesis. To this extent, the results related to these optimisations are also reported and discussed.

The remainder of this chapter is organised as follows, Section 7.1 re-iterates the research questions and how they relate to each of the evaluation areas. Followed by Section 7.2 which explains the experimental setup such as implementation, datasets and experiment parameters. Then Section 7.3 summarises the common evaluation criteria used throughout the chapter, followed by Section 7.4 which presents the results and discussion concerned with the distributed processing of RDF in the cloud. Subsequently, Section 7.5 evaluates the dictionary encoding approach and presents the results and findings, followed by the evaluation of the forward rule-based reasoning algorithm using Google BigQuery in Section 7.6. Most importantly, feedback on the viability of CloudEx is presented in Section 7.7. Finally, this chapter is concluded in Section 7.8 with a summary of the key findings of the evaluation.

7.1 Research Questions

The evaluation presented in this chapter is centred around the three research questions and the contributions of this thesis. These can be categorised in terms of both CloudEx and ECARF. For CloudEx the following research question and evaluation areas are discussed in this chapter:

- *Q1. How can a cloud-based system efficiently distribute and process tasks with different computational requirements?.* The CloudEx framework is evaluated in the context of RDF processing and the two areas listed below (Dictionary encoding and Forward reasoning). The key areas being evaluated are **Distributed RDF processing** and **CloudEx evaluation** which are covered in Sections 7.4 and 7.7

respectively. The **Distributed RDF processing** area focuses on evaluating the framework's ability to scale and distribute the processing of RDF dataset files. Additionally, this area is a key area as it facilitates the processing of the datasets using the other two areas listed below.

Additionally, the following ECARF related research questions and evaluation areas are also discussed:

- *Q2. How can an efficient dictionary that fits in-memory be created for encoding RDF URIs from strings to integers?*. The dictionary encoding algorithm developed in this thesis is evaluated in terms of performance, memory usage and the size of the generated dictionary. The key area being evaluated is **Dictionary encoding** which is covered in Section 7.5.

- *Q3. How can cloud-based big data services such as Google BigQuery be used to perform RDFS rule-based reasoning?*. The forward RDFS rule-based reasoning using columnar databases is evaluated by utilising Google BigQuery. The key area being evaluated is **Forward reasoning** which is covered in Section 7.6.

7.2 Experiements Setup

This section sets the scene for the evaluation methodology followed in this thesis, firstly the implementation details of both CloudEx and ECARF are provided. Then, the experimental platform setup is explained followed by a brief review of the datasets used for the evaluation. Lastly, the evaluation criteria of the various aspects of CloudEx and ECARF are explained.

7.2.1 Implementation

The CloudEx implementation was explained in detail in Section 5.6, as it was noted, the implementation is separated into two parts, a generic part and a cloud provider specific part. Both parts are open source, publicly available[1] and can be imported as a **J**ava **AR**chive (JAR) [132] library. For the work done in this thesis a Google Cloud Platform (GCP) specific implementation was developed which utilises the following GCP components:

- Google Compute Engine (GCE) for creating and managing processors,
- Google Cloud Storage (GCS) for storing RDF dataset files and as a shared storage between the coordinator and processors, and
- Google BigQuery for providing fast SQL-like queries on the *triple-table*.

Additionally, a prototype for the ECARF framework was developed using the Java programming language version 1.7. This prototype imports the CloudEx implementation as a library and defines a number of user-defined tasks (Section 5.6.3) for RDF datasets processing, which are summarised in Appendix A.1. Some of these tasks, mainly the ones executed by the processors are implemented in a multi-threaded manner to benefit from multi-core virtual machine types. An end-to-end CloudEx job was defined to execute these tasks, the JSON definition of this job is presented in Appendix A.2. These tasks fall into one of the three areas mentioned previously and evaluated in this chapter, namely distributed RDF processing, dictionary encoding and forward reasoning.

Additionally, the NxParser [133] open source library was utilised to parse the datasets that are in N-Triples format. Moreover, for the dictionary algorithms Java collection

[1] `https://cloudex.io/`

structures such as HashSets and HashMaps are serialised directly to disk. In this regards, the Kryo [134] Java serialisation library is utilised due to its improved performance compared to the Java built-in serialisation support, as demonstrated by the JVM serialisation benchmark report [135]. Furthermore, ECARF extends the cloudex-core's `CloudService` component with additional API wrappers for Google BigQuery. Similar to the CloudEx implementation, the ECARF implementation is open source and is publicly[2] available.

7.2.2 Platform Setup

To conduct the experiments on the Google Cloud Platform (GCP), a new *project* was created in the *us-central1-a* zone using the Google Cloud Console. Then, under this *project* a new VM image (*vmi*) was created based on the Linux Debian 8.3 distribution, which includes the prototype for the ECARF framework and the Java runtime as explained in Section 5.6.5. Subsequently, the various Gzip-compressed datasets files are uploaded into new Cloud Storage buckets (Section 3.4.2), one bucket for each of the datasets. The coordinator is then started and provided with the filename of the JSON definition of the job to execute, as summarised in Sections 5.6.6 and 6.4.

7.2.2.1 Virtual Machine and Disk Types

The configuration for each job, such as job data (Section 5.4.1) and virtual machine configurations (Section 5.4.2), are controlled by changing the job JSON definition file. This provides the flexibility of changing the experiment parameters without changing the task classes that implement the algorithms being evaluated. For example, when utilising a VM with high memory configurations, the Java memory available to the ECARF prototype is changed using the startup script in the job JSON definition (as

[2] `http://ecarf.io`

shown in Appendix A.2).

A range of GCE virtual machine types [96] have been utilised for the CloudEx processors, primarily the standard and high-memory machine types (Section 3.4.1). This chapter experiments with 1, 2, 4, 8 and 16 processors respectively and utilises VMs with multiple cores up to 16 cores. The default machine type utilised is the *n1-standard-2* type, which has 2 virtual CPUs and 7.5 GB of memory. The evaluation have experimented with both standard and Solid State Disks (SSD) with two sizes, 10 GB and 100 GB to determine their impact on the performance of the algorithms. However, it was found that changing the disk type or size does not have any impact on the performance of the algorithms, primarily due to all the algorithms either being processor bound or network bound, in particular when downloading and uploading files from and to Cloud Storage.

7.2.3 Results Gathering

As noted in Chapter 5, that CloudEx processors are ephemeral VMs which are deleted once the job execution is done. To facilitate the results gathering from these processors before they get deleted, a task `DoUploadOutputLogTask` has been created which is run at the end of each job. This task uploads the processor's log file to a Cloud Storage bucket after prefixing it with the processor's name. Each of the log files contains detailed execution times and memory usage of each task or any particular algorithm that is of interest to the evaluation. Initially, these logs files were parsed manually to extract the needed data, however this task had become tedious specially when using a large number of processors. For this reason a utility was developed to automatically download all the log files from a particular Cloud Storage folder and extract the relevant results in CSV format.

Table 7.1: Evaluation Datasets

Dataset	Raw Size (GB)	Gzip Size (GB)	No. of Triples	No. of Files
Swetodblp	4.6	0.16	14,936,600	8
DBpedia	44.6	4.9	322,728,835	66
LUBM	196.6	6.8	1,092,030,000	45

7.2.4 Datasets

To evaluate the algorithms developed in this thesis, a mix of real-world and synthetic datasets are utilised, these are summarised in Table 7.1. These datasets are publicly available and have also been used in other related research, hence facilitating direct comparison with other results where possible. It is worth noting that the datasets used are in N-Triples [35] format. Datasets that are in RDF/XML format are converted to N-Triples using Apache Jena command-line tools[3] which can convert datasets to and from a variety of formats. The following is a brief description of each of the datasets:

- **SwetoDblp**[4] [136], is a collection of computer science bibliography data. This dataset is relatively small and helps to evaluate the BigQuery forward reasoning optimisations without incurring huge cost.

- **DBpedia**[5] [8], is a collection of structured data extracted from Wikipedia, the English version of the 3.9 DBpedia dataset has been used. This dataset contains a variety of URIRefs and files of various sizes and compression ratios, some of which contain huge amounts of text, more specifically the whole contents of Wikipedia pages. This dataset is utilised to determine the partitioning parameter for the distributed RDF processing and to evaluate the dictionary encoding algorithm.

[3] `https://jena.apache.org/documentation/io/rdf-output.html`
[4] `http://datahub.io/dataset/sweto-dblp`
[5] `http://wiki.dbpedia.org/Datasets`

- **LUBM** [66], is an artificial benchmark tool that can be used to generate an arbitrary number of triples. The LUBM generator generates a university domain dataset with a variable number of universities and their related entities. LUBM 8k (8000 universities) was generated. This dataset is contained in files of equal sizes, contains uniform URIRefs and does not have long text like the DBpedia dataset.

The schema data for all the datasets are provided in separate files, in effect, the closure of the schema data is computed under the *ρdf* semantics (Section 2.1.4) using Jena. The materialised schema data was then uploaded to the same Cloud Storage bucket as the dataset. Each Cloud Storage bucket was created in the same region where the Compute Engine VMs are launched, which according to the documentations, provides the best file upload / download performance [137].

7.3 Common Evaluation Criteria

This section summarises the common evaluation criteria for the algorithms developed in this thesis, in order to determine their feasibility and facilitate comparison with other related work. Other specific evaluation criteria are summarised in their relevant sections below — namely Sections 7.4, 7.5, 7.6 and 7.7 — for each of the three main areas summarised in Section 7.1.

7.3.1 Runtime and Scalability

Runtime is one of the key aspects to measure when evaluating the algorithms developed in this research. Runtime is the total time taken to complete a particular job or task that processes a particular input on a number of processors. The scalability of the algorithms presented in this thesis is evaluated by changing the number of processors

and measuring the resultant runtime. From these values, the speedup and efficiency of the algorithms can be calculated. The speedup s_n of an algorithm executed on n processors is calculated by dividing the runtime t_1 when using one processor by the runtime t_n when using n processors as follow:

$$s_n = \frac{t_1}{t_n} \tag{7.1}$$

The efficiency e_n when using n processors is calculated as follows:

$$e_n = \frac{s_n}{n} = \frac{t_1}{t_n \times n} \tag{7.2}$$

A linear speedup means that $s_n = n$ and $e_n = 1$ for all n.

7.3.2 Cost of Computing Resources

Cost is one of the key aspects with respect to using cloud computing resources. Consumer can have access to a larger number of computing resources compared to physical computing clusters with fixed resources. Having said this, many of the existing research reports on runtime rather than cost, furthermore, the cost is likely to vary between cloud providers. However, the evaluation measures and reports the cost involved in executing the various RDF processing jobs, in order to use the cost alongside the runtime as an indication of improvement when evaluating a number of approaches for the same algorithm.

7.3.3 Multiple CPU Cores

Cloud computing enables the provision of virtual machines with multiple CPU cores, additionally cloud providers maintain a CPU to memory ratio. Thus, when acquiring a VM with a larger memory, it is likely that such VM will have more CPU cores than

a VM with less memory. Consequently, the additional CPU cores can be utilised by implementing the user-defined CloudEx tasks in a multi-threaded manner. Thereby the evaluation also measures the impact on runtime and cost when utilising more CPU cores.

7.3.4 Triples Throughput

The forward reasoning algorithm utilised, uses a number of rules to infer new knowledge in the form of new statements or triples. The reasoning process will consume a number of input triples and progress to infer new triples. Inline with other published results [19, 64, 21], the rate at which the input triple are consumed per second is measured by using a *Ktps* unit, to denote "thousand triples per second". This is calculated as:

$$throughput(Ktps) = \frac{input\_triples}{runtime}$$

7.4 Distributed RDF Processing

This section evaluates the ability to distribute the processing of RDF datasets between a number of processors. Mainly, it evaluates the scalability of two processor tasks `ExtractCountTerms2PartTask` and `ProcessLoadTask` (Appendix A.1). Despite the fact that the `AssembleDictionaryTask` is executed between these two tasks, it is not a distributed task, hence is executed in serial and is evaluated separately in Section 7.5. Firstly, the DBpedia dataset is used to experiment with and report on the partitioning factors that results in a balanced workload between the processors. Additionally, this section reports on the runtime and scalability when processing the LUBM dataset both on multiple processors (1 to 16) and on a single processor with multiple cores

(up to 16). This section also evaluates the total time required to preprocess the dataset files and load them into BigQuery ready for forward reasoning. This load time is reported by a number of related work facilitating direct comparison. The tasks required to preprocess and load the dataset into BigQuery include all the tasks shown in the job definition in Appendix A.2 up to the `LoadBigDataFilesTask` task.

7.4.1 Partitioning Factor

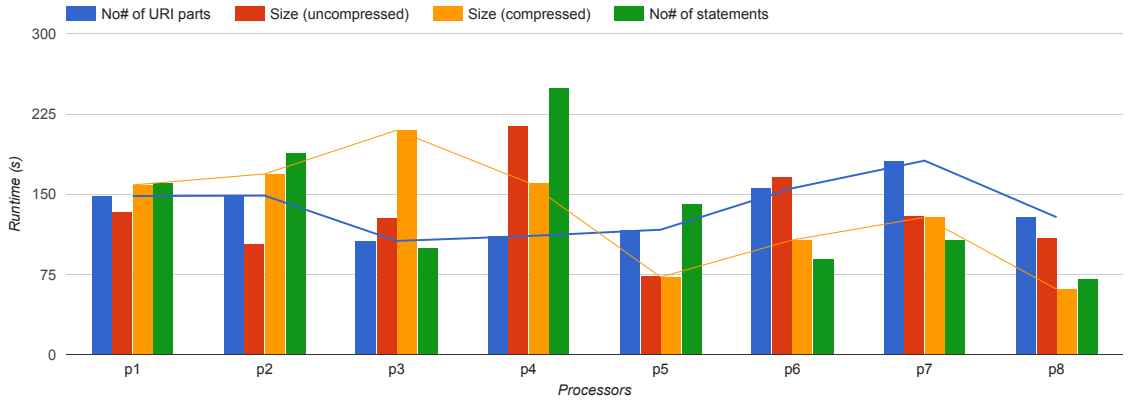
As noted in Section 5.3.1, the CloudEx workload distribution approach is based on the Bin Packing algorithm, for this reason it is needed to determine the partitioning factors to use for each of the dataset files. These factors can be summarised as follows:

- File size, compressed and uncompressed.
- Number of triple statements in the file.
- Number of unique URI parts in the file that is extracted for the dictionary, this is discussed in Section 7.5.

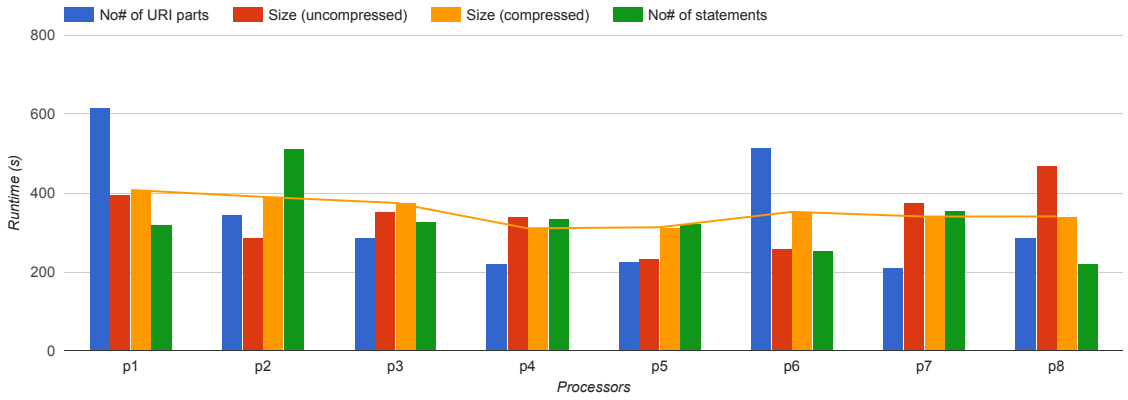
Figure 7.1, shows the runtime for 8 processors on the DBpedia dataset when using each of the partitioning factors. The DBpedia dataset has been used for this evaluation because the dataset has files with a wide range of sizes with varying contents and compression ratios. For this reason, the dataset provides a worst case scenario for the distributed processing when compared with more uniform datasets like Swetodblp and LUBM.

7.4.1.1 ExtractCountTerms2PartTask

As seen in Figure 7.1(a), for this task the processors have different runtimes for all the partitioning factors. This indicates that there are multiple factors affecting the load



(a) ExtractCountTerms2PartTask



(b) ProcessLoadTask

Figure 7.1: Partitioning factor on DBpedia using 8 n1-highmem-4 processors (4 cores, 26 GB RAM).

balancing between the processors. Despite this, it is clear that the number of URI parts (Blue) provide the optimum partitioning strategy compared to the rest, because, as noted in Section 5.1.3, the coordinator waits for all processors to finish before proceeding. However, the number of URI parts is not known before the execution of this task, because this task is responsible for extracting these parts. Consequently, the number of URI parts can not be used as a partitioning factor. The second best partitioning factor is the compressed file size shown in Orange, this factor is known in advance and can be used to partition the workload for this task.

The reason why the number of URIs parts strategy is not balanced between the processors was investigated. It was found that the slowest processor $p7$ is assigned one file that has the largest number of URI parts, clearly this can not be partitioned further. This is one of the issues that faces CloudEx's Bin Packing approach, particularly when an item is excessively larger than others leading to load balancing issues when assigned to a processor. On the other hand, the second slowest processor *p6*, although assigned the lowest number of URI parts, it had the most number of files and hence the majority of the time was spent on downloading these files. This further strengthens the fact that with diverse datasets like DBpedia, the workload partitioning for this task needs to take into account more that one factor.

It is also worth noting that for datasets with a variety of URIRefs this task is memory intensive due to the fact that a HashSet is kept in memory for each of the files. Additionally, as the files are downloaded from Cloud Storage before being processed the task is also bound by the time required to download these files.

7.4.1.2 ProcessLoadTask

Figure 7.1(b) shows the runtime for the `ProcessLoadTask`, as seen the compressed size provides the optimum partitioning strategy compared to the rest. Additionally, in contrast to `ExtractCountTerms2PartTask`, for this task the processors have runtimes that are largely balanced. This is due to the fact that this task is multi-threaded and CPU intensive, as each input file is decompressed on the fly then each triple statement is encoded, compressed on the fly and written to an output file. In this case a larger compressed file will take longer to process than a smaller one, hence the compressed file size seems to be the best partitioning factor for this task. It is also worth noting that the input files are available locally on disk from the execution of the `ExtractCountTerms2PartTask`, once an input file is processed the output file is

Table 7.2: Partitioning Factor runtime (seconds) comparison for DBpedia using 8 n1-highmem-4 processors.

		ExtractCount Terms2PartTask		**ProcessLoadTask**		
Part. Factor	**Node**	**Average**	**STD.**	**Average**	**STD.**	**Total**
Size	Coordinator	236	3	502	22	738
	Processor	132	40	345	77	
Size (gzip)	Coordinator	233	9	440	39	673
	Processor	133	50	354	41	
No. of statements	Coordinator	274	5	541	15	814
	Processor	139	57	331	83	
URI Parts	Coordinator	209	12	635	49	844
	Processor	137	25	339	145	

uploaded to Cloud Storage.

7.4.1.3 Partitioning Factor Summary

Table 7.2 shows the average runtime in seconds for each of the two tasks evaluated in this section both for the processors and coordinator. As discussed in the previous two sections, the URIs parts factor provides the best partitioning strategy for the `ExtractCountTerms2PartTask`. This is evident from Table 7.2 as it has the lowest standard deviation between the processors average runtime, moreover, it has the shortest runtime reported by the coordinator. On the other hand, for the `ProcessLoadTask`, the compressed file size provides the best partitioning factor. Evidently, it provides the shortest time reported by the coordinator and lowest standard deviation compared to the other factors. As a whole, the compressed file strategy provides the best partitioning factor, mainly due to the fact that the `ProcessLoadTask` has a longer runtime than the `ExtractCountTerms2PartTask`. Any optimisations on the earlier task will result in a shorter runtime for the two tasks combined.

Table 7.3: LUBM 8K coordinator (C) and processors (P) runtime (seconds), speedup (Sp.) and efficiency (Eff.) for ExtractCountTerms2PartTask and ProcessLoadTask on up to 16 n1-standard-2 processors.

		ExtractCountTerms2PartTask				ProcessLoadTask						
No. of Proc.	Acq. Time	C	P	Sp.	Eff.	C	P	Sp.	Eff.	End to End	Sp.	Eff.
1	66.24	2137	2118	1.00	1.00	4547	4534	1.00	1.00	6820	1.00	1.00
2	72.92	1091	1067	1.99	0.99	2338	2297	1.97	0.99	3566	1.91	0.96
4	75.68	573	547	3.87	0.97	1213	1170	3.88	0.97	1940	3.52	0.88
8	86.92	320	277	7.65	0.96	687	603	7.52	0.94	1178	5.79	0.72
16	116.08	226	148	14.31	0.89	403	321	14.12	0.88	854	7.98	0.50

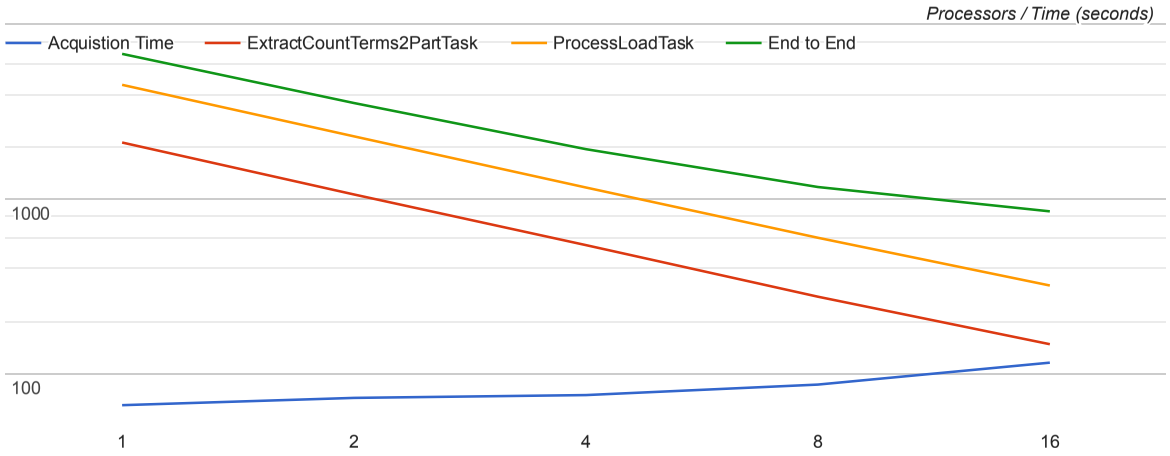


Figure 7.2: LUBM 8K end to end and processors runtime (log-scale) vs. number of processors for ExtractCountTerms2PartTask and ProcessLoadTask on up to 16 n1-standard-2 processors.

7.4.2 Horizontal Scalability

This section evaluates the scalability of the `ExtractCountTerms2PartTask` and `ProcessLoadTask` in terms of increasing the number of processor and measuring the runtime, then the speedup and efficiency are calculated using Formulas 7.1 and 7.2. For these experiments the LUBM dataset was used, which provides approximately the same number of statements in each file. Therefore, the dataset files can be divided equally between processors using the compressed file size as the partitioning factor. Table 7.3 shows the results of distributing the processing of the LUBM dataset files between 1 to 16 processors. The table shows the average runtime for the `ExtractCountTerms2PartTask` and `ProcessLoadTask` for both the coordinator and the processors. Additionally, for the processors the table shows the speedup and efficiency for both of the tasks in addition to the end to end runtime, speedup and efficiency. The end to end runtime includes serial tasks such as the time to acquire the processors — *acquisition time* t_{aq} (Section 5.5.1.2) — and the time required to assemble the dictionary (`AssembleDictionaryTask`), which takes 35 seconds on average for LUBM.

As can be seen from Table 7.3 and Figure 7.2, the runtime for the tasks shows almost linear speedup in term of the processors runtime. The runtime reported by the coordinator for each of the tasks includes a slight overhead to update the metadata for the processors and waits for them to update their metadata to indicate their availability. The average time required to update the metadata is between 4 to 6 seconds, and the total metadata update time required for any of the task is on average 20 seconds. This is the time needed for the coordinator to update the processor metadata, for the processor to read and update its metadata before executing the task, then again once the task is completed and finally the time needed for the coordinator to reads the metadata.

Figure 7.2 shows that from the coordinator's point of view there is a small improvement of the overall runtime when increasing the processors from 8 to 16 when compared to the runtime reported by the processors. As mentioned previously, this is due to the processors coordination overhead due to the metadata updates, which becomes noticeable when distributing an input that is not proportionally large between a large number of processors. Additionally, the end to end speedup is not linear as it includes tasks that are executed serially as noted earlier, in addition to the added overhead of the metadata updates throughout the tasks execution. Furthermore, the *acquisition time* increases slightly when the number of processor increase, this is due to the implementation nature of CloudEx, which will be discussed in Section 7.7.

7.4.3 Comparison of LUBM 8K Load

The LUBM dataset is used by many Semantic Web approaches to evaluate their RDF processing algorithms such as loading, dictionary encoding and forward reasoning. The loading time for LUBM 8K is commonly reported by related work, this is usually the time taken to load and encode the dataset files ready for the forward reasoning process. For some approaches this is the time required to import the dataset into a database. The load time reported in this section is based on two stages, the first is the preprocessing done by the three tasks `ExtractCountTerms2PartTask`, `AssembleDictionaryTask` and `ProcessLoadTask`. The second stage is the time taken by BigQuery to load the dataset which is on average 4 minutes 26 seconds for LUBM 8K.

Table 7.4 provides a comparison of this approach with other related work that loads LUBM 8k, apart from [81] that load LUBM 10k with only 200 M triples difference. Since their approach exhibits linear scalability in terms of input, the time required to load LUBM 8k will be around 23 minutes. The approach developed in this thesis

Table 7.4: LUBM dataset load runtime comparison

LUBM	**Work**	**Load (h:m:s)**	**Hardware**
8k - 1.1 B	Bishop et al. [56]	14:00:00	4-core 2.93 GHz PC with 12 GB RAM
8k - 1.1 B	Urbani et al. [67]	01:10:30	32 x 2-core 2.4 GHz 4 GB RAM Hadoop cluster
8k - 1.1 B	Kolovski et al. [138, 58]	00:28:11	64-core 3.0 GHz 512 GB RAM SPARC Oracle RDBMS server
10k - 1.3 B	Weaver. J [81]	00:27:11	64-core Opteron 512 GB RAM supercomputer
8k - 1.1 B	This approach	00:18:40	16 x n1-standard-2 2-core 7.5 GB RAM cloud VMs

loads the LUBM dataset in 18 minutes 40 seconds, this includes the time taken by BigQuery to load the encoded dataset (on average 4 minutes and 26 seconds). It is clear that this approach is able to load the dataset in less time without utilising any powerful or dedicated hardware, in fact, given that the hourly cost of the *n1-standard-2* VM is \$0.10, the total cost incurred by this approach to load the LUBM dataset is $\frac{0.10}{60} \times 16 \times 18.68 = \$0.50$. As can be seen the approach presented in this thesis provides a better load time with no upfront investment and on a pay-as-you-go approach. This approach is able to provide a better load time by utilising an efficient dictionary encoding process and splitting the load process into a number of optimised serial and parallel tasks.

7.4.4 Vertical Scalability

Cloud VMs have a CPU cores to memory ratio, hence applications that have high memory requirement can also benefit from the added CPU cores by utilising parallel execution using multi-threading. This section reports on the performance gains achieved when using multi-threaded tasks for the RDF dataset processing, namely `ExtractCountTerms2PartTask` and `ProcessLoadTask`. Table 7.5 shows the runtime

Table 7.5: LUBM 8K load runtime, speedup and efficiency on single 1n-standard-1 to n1-standard-16 processor.

No. of Cores	Memory (GB)	End to End (s)	Speedup	Efficiency
1	3.75	8589	1.00	1.00
2	7.5	6822	1.26	0.63
4	15	3903	2.20	0.55
8	30	2141	4.01	0.50
16	60	1474	5.83	0.36

and scalability results for the LUBM dataset when using one processor with variable CPU cores, ranging from 1 to 16. As seen, by using multiple execution threads to utilise the available CPU cores, a level of parallelism on a single processor can be achieved. This is useful with tasks that require shared memory access such as the `AssembleDictionaryTask`. Moreover, as evident from Table 7.5, adding more CPU core, which in turns means more memory, improves the overall runtime. However, the speedup is not linear due to many factors such as the contention of threads over shared resources such as the network bandwidth and disk access.

7.5 Dictionary Encoding

The dictionary encoding is carried out in three steps, firstly the `ExtractCountTerms-2PartTask` extracts all the URI parts for the dictionary keys, secondly the dictionary is assembled using the `AssembleDictionaryTask`, thirdly the `ProcessLoadTask` uses the dictionary to encode the dataset. The `AssembleDictionaryTask` is implemented in a multi-threaded manner and is executed on one processor with a memory size proportionate to the size of the extracted URI parts files. The dictionary assembly evaluation uses all the three datasets summarised in Section 7.2.4 and experiments with the three strategies presented in Section 6.2.3 for splitting the URIRefs into two parts before encoding.

Table 7.6: Comparison of Standard, Hostname + first path part (1st PP) and Hostname dictionary split strategies for Swetodblp, DBpedia and LUBM datasets.

	Swetodblp			DBpedia			LUBM		
	Stand.	Host. +1stPP	Host.	Stand.	Host. +1stPP	Host.	Stand.	Host. +1stPP	Host.
Extract Files Size (MB)	81	103	104	4,031	4,118	4,610	317	5,285	14
No. Extract URI Part	3,738,132	3,788,657	3,791,842	165,119,090	166,856,992	176,000,886	5,857,835	109,218,289	282,937
No. Dict. Keys	2,494,280	2,654,293	2,655,787	53,551,014	55,784,063	66,128,766	5,769,284	109,168,015	169,646
Dictionary Size (MB)	64	97	103	1,760	1,877	2,537	314	6,325	6
Max Mem. (GB)	0.5	0.5	0.5	12	12	13	1	31	0.5
Assemble Time (s)	6	6	6	510	600	820	36	889	8

Table 7.7: Dictionary encoding metrics, such as encoded data and dictionary sizes, compression rate and BigQuery scanned bytes improvements for Swetodblp, DBpedia and LUBM.

Dataset	**Size (GB)**	**Raw enc. data (GB)**	**Raw dict. (GB)**	**Enc. data gzip (GB)**	**Dict. gzip (GB)**	**Comp. rate**	**Query scan (GB)**	**Impr. (%)**
Swetodblp	4.62	0.88	0.06	0.13	0.02	29.93	0.27	94.08
DBpedia	47	13.8	1.72	4.48	0.91	8.72	6.26	86.68
LUBM	169	31.4	0.31	6.16	0.03	27.29	22	86.98

The size of the dictionary is evaluated for each of these strategies, additionally, if the dictionary assembly process is memory intensive this section also reports the memory usage pattern. Moreover, this section shows the BigQuery savings achieved by the dictionary encoding in terms of scanned data, which is the primary cost factor when using BigQuery. Finally, the dataset compression rate and dictionary sizes are evaluated against other large-scale distributed RDF dictionary encoding. The compression rate is calculated similar to [67] as follows:

$$compressionRate = \frac{originalDatasetSize}{gzippedEncodedDatasetSize + gzippedDictionarySize}$$

7.5.1 URIRefs Split Strategy

Table 7.6 provides a comparison of the three URIRefs split strategies (summarised in Section 6.2.3), additionally Figure 7.3 shows the size of the dictionary for each of the three strategies. As can be seen, the *Standard* split strategy provides the smallest dictionary for real-world datasets (Swetodblp, DBpedia) compared to the other two split strategies and consequently requires the least memory usage (as evident from the DBpedia dataset). In contrast, the *Hostname* strategy provides the best dictionary size for the LUBM dataset, unsurprisingly, due to the uniformity of this synthetic dataset. The *Hostname* strategy, which provides the largest dictionary for both Swetodblp and DBpedia, seems to provide a very small dictionary for LUBM of only 6 MB. This shows that each dataset has its own characteristics and the split strategy that provides the smallest dictionary, can be different from one dataset to another.

Additionally, as shown in Figure 7.3 the *Standard* split strategy provides the second best dictionary size for LUBM at 314 MB. On the other hand, the *Hostname + 1stPP* is the worst strategy for LUBM, resulting in a very large dictionary with a size of 6.1 GB. Although an n1-highmem-4 VM with 26 GB of memory was enough to

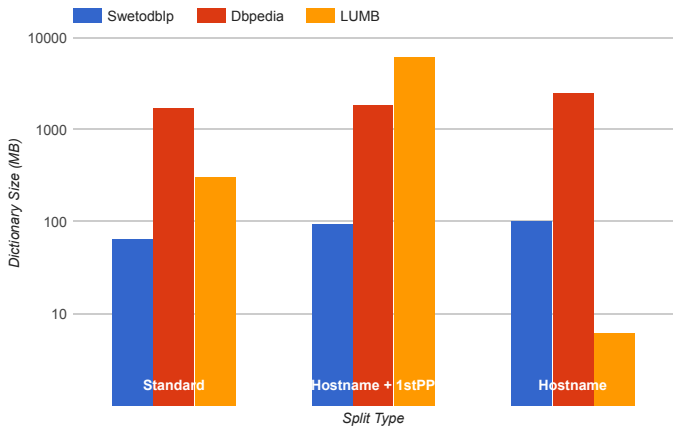


Figure 7.3: Comparison of dictionary size (logscale) when using Standard, Hostname + first path part (1stPP) and Hostname split strategies.

assemble the DBpedia dictionary, the assembly of this large LUBM dictionary needed an n1-highmem-8 VM with 52 GB of memory.

Despite the fact that VM specification — in terms of memory — were set manually to assemble the dictionary for both DBpedia and the LUBM worst case (*Hostname + 1stPP*), the VM specification can be automatically set. As shown in Table 7.6, the dictionary size is dependent on the number of dictionary keys, which in turn is proportional to the sizes of the extract files generated by `ExtractCountTerms2PartTask`. Based on these parameters, the memory requirements for the dictionary assembly can be estimated before hand, although the efficiency of this estimation is likely to be impacted by the fact that a number of URI parts can be repeated in many of the extract files. For example, the DBpedia extract files for the *Standard* strategy contain 165,119,090 URI parts of which only 53,551,014 are unique.

7.5.2 Dictionary Assembly Memory Footprint

In terms of the memory footprint of the `AssembleDictionaryTask`, the evaluation has analysed the usage of the memory intensive dictionaries for both the DBpedia

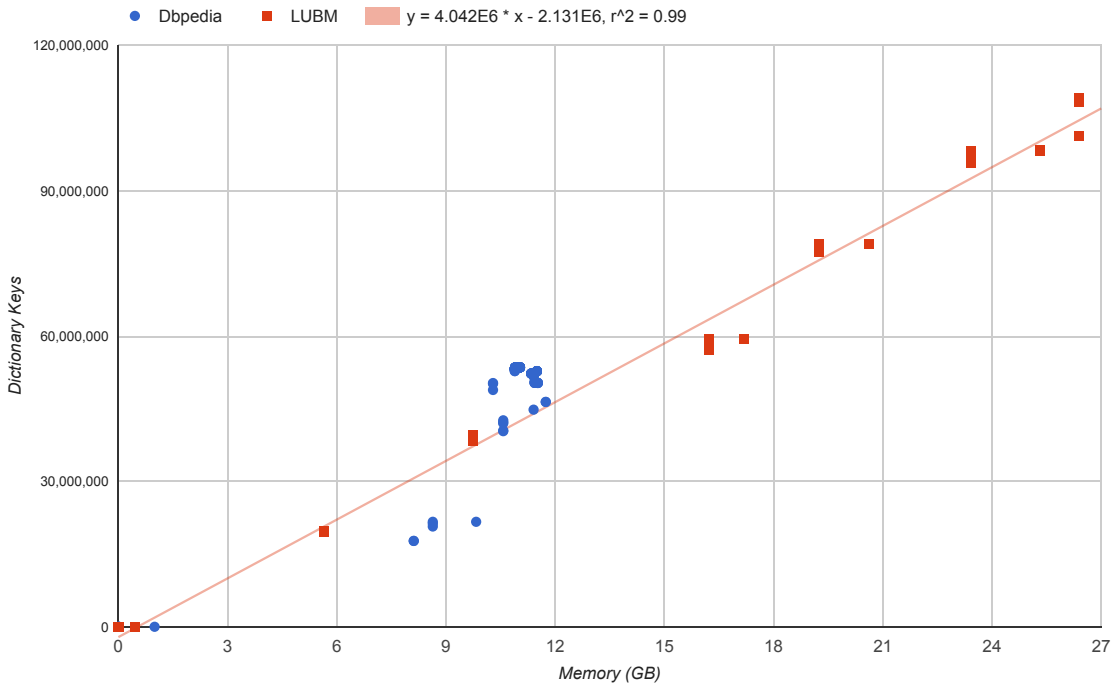


Figure 7.4: Dictionary assembly memory usage for DBpedia and LUBM datasets.

and LUBM datasets. As shown in Table 7.6, these include the three strategies for the DBpedia dataset and the *Hostname + 1stPP* for strategy LUBM. The memory usage patterns are shown in Figure 7.4 for both datasets with a regression line showing the relationship between the number of dictionary keys and the dictionary assembly memory usage.

It is worth noting that the maximum allocated Java Virtual Machine (JVM) memory needs to be more than the memory requirements for the dictionary assembly. For example for DBpedia dataset the dictionary required 12GB of memory, whilst the maximum JVM memory in this case is set to 23GB and the actual VM has a maximum of 26GB of main memory. For the LUBM worst case strategy (*Hostname + 1stPP*), the dictionary memory usage is 31GB, whilst the JVM memory is set to 50GB on a VM with 52GB of main memory. It was observed that if the dictionary assembly uses most of the JVM memory, garbage collection occurs more frequently and the

assembly process slows down considerably. Moreover, if the JVM is assigned more memory than the system can accommodate this can result in the crash of the JVM. From Figure 7.4 the memory usage for the dictionary assembly can be calculated based on the number of dictionary keys using the linear regression formula shown on the figure. For example, given that the maximum memory that can currently be allocated to a VM on the Google Cloud Platform is 208GB (using an n1-highmem-32 VM), this dictionary encoding approach can readily assemble very large dictionaries with more than 450 million keys.

7.5.3 BigQuery Scanned Bytes Improvements

The key motivation behind implementing the dictionary encoding process, is to reduce the size of the total data scanned by BigQuery when conducting forward RDF reasoning and in turn reduce cost. Table 7.7 shows the dictionary encoding metrics such as the original, encoded and gzip encoded data and dictionary sizes for the Swetodblp, DBpedia and LUBM datasets. The "size in GB" column shows the original size of the data which will be scanned by BigQuery, on the other hand, the "query scan in GB" column shows the encoded size that will be scanned by BigQuery. As shown, the dictionary compression achieves considerable improvements and subsequently BigQuery cost reduction. The percentages of these improvements are 94.08%, 86.68% and 86.98% for the Swetodblp, DBpedia and LUMB datasets respectively.

7.5.4 Comparison of Dictionary Encoding

It was noted in Section 7.4.3 that the approach developed in this thesis outperforms the results reported by [138, 56, 58, 67, 81] when loading the LUBM 8K datasets. This is primarily due to the efficient dictionary encoding process, which means the overall dataset load process takes less time as the dictionary can be held in its en-

Table 7.8: Comparison of large scale RDF dictionary encoding.

				Encoded Data size (GB)		Dictionary size (GB)	
Work	Dataset	Size (GB)	Comp. rate	Raw	Comp.	Raw	Comp.
Urbani et al. [67]	LUBM 8k	158.9	10.02	-	14	-	1.9
Weaver. J [81]	LUBM 10k	224	-	29.8	-	23	-
This approach	LUBM 8k	169.6	27.29	31.4	6.16	0.31	0.03
Urbani et al. [67]	DBPedia (110M)	17.4	7.33	-	1.4	-	1
This approach	DBpedia (322M)	47	8.72	13.8	4.48	1.72	0.91

tirety in memory. Table 7.8 provides a comparison of the LUBM 8K and DBpedia datasets compression rate and dictionary sizes presented here against results reported by [67, 81]. In terms of LUBM, the approach of this thesis provides the smallest dictionary and achieves better compression rate. Additionally, for DBpedia dictionary and compression rate reported in this section are better than those reported by Urbani et al. [67], this is considering that this evaluation used a newer version of the dataset which is three times larger than the version used by Urbani et al., despite this, the compressed dictionary size reported here is smaller.

7.6 Forward Reasoning

This section evaluates the RDFS rule-based reasoning approach presented in Section 6.3.3 by utilising Google BigQuery. Mainly, the evaluation is concerned with the `DoReasonTask` processor task — summarised in Appendix A — by considering two factors for the evaluation, firstly cost followed by runtime. BigQuery charges per the size of data scanned, for larger datasets this cost can outweigh the cost of the virtual machines used for the reasoning process. For this reason, this section reports on the improvements achieved by a number of optimisations to reduce both the size of the

data scanned by BigQuery and the time taken to retrieve the query results, which in turns improves the overall runtime.

Initially, the Swetodblp dataset was used to report on the improvements achieved by these optimisations. Then, forward reasoning is performed on both the DBpedia and LUBM datasets by using a number of processors up to 16, additionally, the evaluation also experiments with a single multi-core processor. When distributing the reasoning process between a number of processors, the relevant term partitioning strategy is used, consequently its effectiveness is also evaluated. For the experiments conducted in this section, the evaluation reports the total reasoning time and the reasoning throughput in terms of thousands triples consumed per second (*Ktps*) as explained in Section 7.3.4. Both the runtime and the *Ktps* are used to compare the results presented here with other related work.

7.6.1 Forward Reasoning Optimisations

BigQuery is the biggest cost factor in the evaluation presented here, this is because if individual queries are issued for each of the productive terms, each query either scans one or two columns for each of the productive terms. For a bigger dataset like DBpedia — where there are 1802 productive terms — this approach could be very costly. For example, issuing 1802 term queries for DBpedia will scan almost 79 TB of data, which amounts to almost $395!. To avoid incurring massive costs the small Swetodblp dataset has been utilised to experiment with a number of approaches in order to reduce the BigQuery cost and the data download times. These approaches can be summarised as follows:

1. **Multiple Queries**: this approach issues an individual query for each of the productive schema terms.

2. **Single Query (SQ)**: this approach issues a single query for all the productive schema terms per processor.

3. **SQ Encoded**: this approach builds further on the Single Query approach and uses encoded data as per the dictionary encoding approach.

4. **SQ Encoded + Load Export**: this approach builds on the previous approach and instead of retrieving the data from BigQuery in batches, the data is exported in a single export. Since the `DoReasonTask` is multi-threaded, the number of files created in the export are based on the number of CPU cores in each of the VMs to facilitate the parallel processing of these files.

5. **Duplicates Removed**: this approach builds on the previous approach and removes the duplicates by utilising an in-memory cache that efficiently stores generated triples so that they are not created again during the reasoning process.

The results for these five approaches are shown in Table 7.9 and Figure 7.5. The results show the number of BigQuery scanned bytes, the data download times from BigQuery, the total reason task time, the number of inferred triples and the cost in USD. As can be seen from table that the *Multiple Queries* approach scans 127 GB (130,959 MB) of data and takes 240 seconds to download the data to the processor. Whereas, the *Single Query* approach only scans 4.78 GB of data, but the data download takes 645 seconds which is around 6 minutes more than the *Multiple Queries* approach. This increase in the download time is due to the size of the downloaded data, because all the queries are joined, the algorithm can no longer just select the subject or both the subject and object columns (Table 6.4) as was the case with the multiple queries. A single query needs to select all the three columns subject, predicate and object to be able to conduct the rule-based reasoning, hence resulting in more data to transfer, longer download and reasoning times.

Table 7.9: BigQuery reasoning improvements comparison for Swetodblp.

	Multiple Queries	Single Query (SQ)	SQ Enc.	SQ Enc. + Load Export	Duplicates Removed
Scanned bytes (MB)	130,959	4,895	568	568	371
Data download time (s)	240	645	426	176	125
Total reason time (s)	756	1085	653	399	392
Inferred Triples	16,964,056	18,032,561	12,590,818	12,590,818	3,951,230
Cost (USD)	0.660	0.054	0.021	0.014	0.013

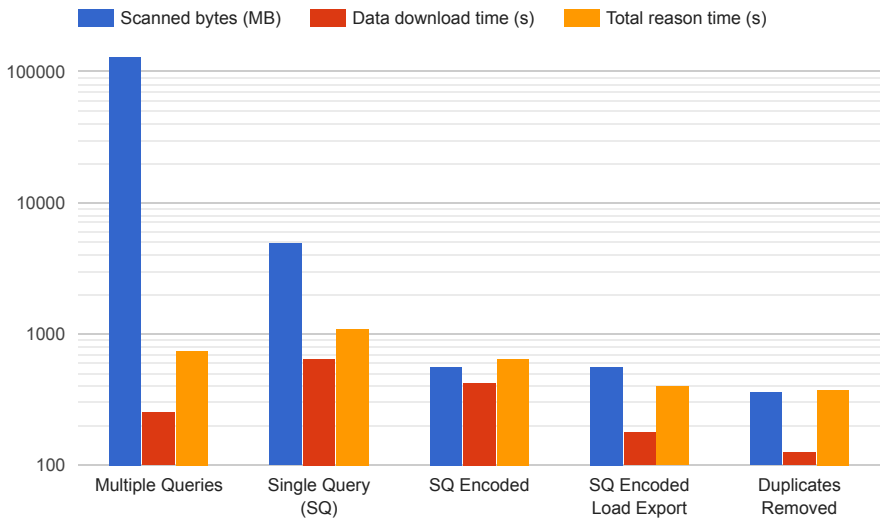


Figure 7.5: BigQuery reasoning improvements comparison for SwetoDblp.

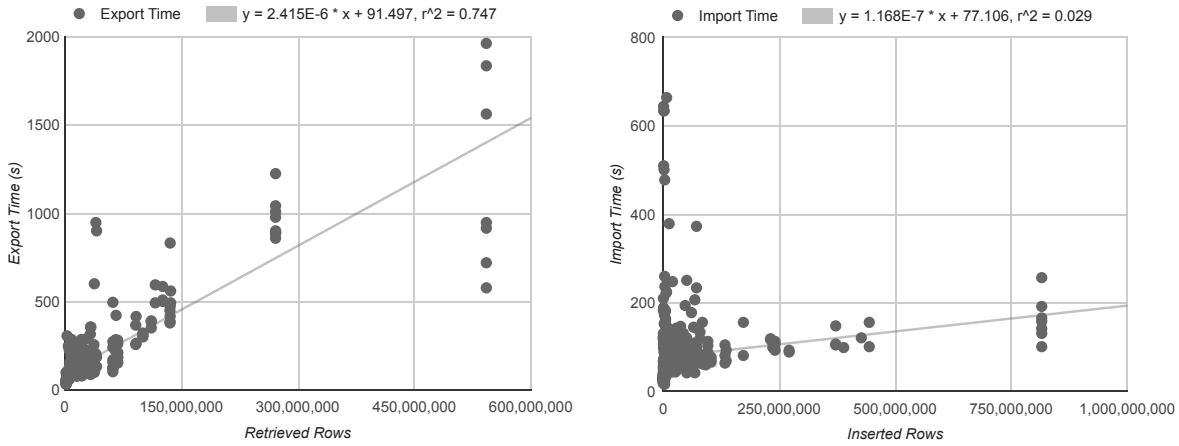
With the *SQ Encoded* approach it can be seen that the scanned bytes, the download times and the total reasoning times are improved and are better than the previous two approaches. However, it was noticed in this case that using the API to retrieve the query results in batches from BigQuery, takes on average 4 seconds for every 100,000 rows, which results in longer download times for large results. Consequently, the evaluation tried to export the query results to Cloud Storage and then downloads the results files to the processor from Cloud Storage. This *SQ Encoded + Load Export* further achieves better download and reasoning times than the previous strategies as shown in both Table 7.9 and Figure 7.5.

Finally, as seen from Table 7.9 that inferred triples contain many duplicates that

varies between the approaches. For this reason the evaluation has experimented with an in-memory cache to store the triples that have already been inferred to avoid inferring them again. This approach is possible due to the small size of the Swetodblp dataset, but is prohibitive for larger datasets like DBpedia and LUBM due to their size and the inability to hold all the inferred triples in memory. The *Duplicates Removed* approach reduces the number of inferred triples considerably, since no duplicates are generated and consequently reduces the query results and the download times. Although the duplicates issue with larger datasets are not addressed in this thesis, these result clearly indicate that dataset can benefit considerably from removing duplicates generated during the reasoning process.

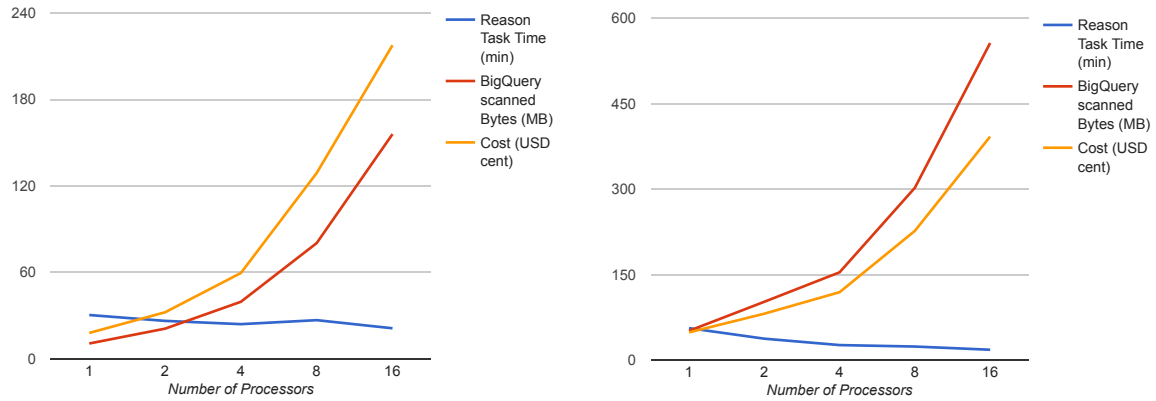
7.6.2 Results of Performing Forward Reasoning

Following on from the improvements discussed in the previous section, the evaluation has utilised the *SQ Encoded + Load Export* approach and ran forward reasoning experiments on both the DBpedia and LUBM datasets. This is achieved by using a number of processors up to 16, additionally, the evaluation experimented with a single multi-core processor. Tables 7.10 and 7.11 show the results of executing the `DoReasonTask` on 1, 2, 4, 8 and 16 processors for the LUBM and DBpedia datasets respectively. Figure 7.7 shows some of these results, in particular the reason task runtime, the number of BigQuery scanned bytes and the overall cost of the forward reasoning process. The results of running the reasoning process on a single multi-core processor are shown in Table 7.12 for both the LUBM and DBpedia datasets. Finally, Figure 7.6 shows the BigQuery import and export times versus the number of retrieved and inserted rows respectively. In the following sections these results are analysed and discussed.



(a) BigQuery export time (b) BigQuery import time

Figure 7.6: BigQuery export and import times vs. retrieved / inserted rows for the DBpedia and LUBM datasets.



(a) DBpedia (b) LUBM

Figure 7.7: Reason task time (mins), BigQuery scanned bytes (GB) and Cost (USD cent) for forward reasoning on DBpedia and LUBM using BigQuery.

7.6.3 Runtime, Load Balancing and Cost

As seen from Tables 7.10 and 7.11, the reasoning for DBpedia takes longer than LUBM when using four or more processors. This is due to the fact that DBpedia has a larger

Table 7.10: BigQuery reasoning results for LUBM.

No. of Processors	Reason Task Time m:s (s)	Processor Total Runtime (s)	Runtime Std. (s)	Processor Reason Runtime (s)	BigQuery scanned Bytes (GB)	BigQuery Data Download (s)	BigQuery Data Insert (s)	Ktps Processor	Ktps End to End	Cost (USD cent)
1	55:50 (3350)	3345	15	861	51.13	2011	171	1,283	329	48
2	37:20 (2240)	2175	64	464	102.14	1171	285	2,377	492	81
4	26:06 (1566)	1098	318	220	153.90	563	136	5,007	720	119
8	23:32 (1412)	741	296	112	302.27	303	119	9,880	814	227
16	17:56 (1076)	588	222	61	556.56	280	105	18,189	1,024	392

Table 7.11: BigQuery reasoning results for DBpedia.

No. of Processors	Reason Task Time m:s (s)	Processor Total Runtime (s)	Runtime Std. (s)	Processor Reason Runtime (s)	BigQuery scanned Bytes (GB)	BigQuery Data Download (s)	BigQuery Data Insert (s)	Ktps Processor	Ktps End to End	Cost (USD cent)
1	30:16 (1816)	1806	153	229	10.49	888	363	1,408	179	18
2	26:12 (1572)	1451	137	146	20.80	606	376	2,228	205	32
4	23:55 (1435)	1130	298	98	39.45	427	307	3,279	225	59
8	26:42 (1602)	1246	400	61	80.20	323	600	5,265	212	129
16	21:04 (1263)	764	179	31	155.86	197	284	10,528	256	218

Table 7.12: LUBM and DBpedia BigQuery reasoning on one n1-standard-8 processor.

Dataset	Reason Task Time (m:s)	Runtime Std. (m:s)	Processor Reason Runtime (s)	BigQuery scanned Bytes (GB)	BigQuery Data Download (s)	BigQuery Data Insert (s)	Ktps Processor	Ktps End to End
LUBM	42:05	09:33	600	51.13	1,348	295	1,894	453
Dbpedia	23:50	01:58	143	10.49	665	399	2,275	227

number of productive schema terms in comparison with LUBM and hence involves more reasoning cycles. On the other hand, the overall runtime generally improves with increasing the number of processors, with the exception of the DBpedia dataset when using 8 processors as shown in Table 7.11, which is primarily due to the BigQuery import taking longer than average (600 seconds) to insert the data. This shows that although the processor reason runtime is reduced with adding more processors, the overall runtime is dependent on the time taken by BigQuery for the export and import of data. Additionally, the overall runtime is impacted by the load balancing of the reasoning process between the of processors. This load balancing issue is evident from the runtime standard deviation which indicates that some processors take longer than others.

7.6.3.1 Load Balancing

The load balancing is impacted by two factors, firstly the BigQuery import and export times, since the BigQuery import or export for one of the processors can take longer than others. Secondly, it was found that although the relevant term partitioning strategy can be used to equally partition the reasoning for the first cycle, it does not necessarily provide equal partitions for the subsequent cycles. This is because newly inferred data can result in some productive schema terms with small number of initial productive instance triples to have a large amount of productive instance triples on subsequent reasoning cycles. Primarily due to the fact that the output of some of the rules in Table 6.4 provide input to other rules. So the load balancing strategy for reasoning needs to consider these relations as done in [139].

7.6.3.2 Cost of Performing Forward Reasoning

In terms of cost, Figure 7.7 shows the overall runtime in minutes, the cost in USD cents and the scanned BigQuery data in Gigabytes. The cost is specifically shown in USD cents and the runtime in minutes to highlight the fact that the cost climbs rapidly with increasing the number of processors due to the increase in the BigQuery scanned data by each processor. This is because each processor will issue a query with all the productive terms assigned to it and such queries will carryout a full scan of the BigQuery table. Since the `DoReasonTask` is multi-threaded, in order to reduce cost the evaluation has experimented with a single multi-core processor for the reasoning process as summarised in Table 7.12. As shown this approach provides a cost effective option in terms of BigQuery scanned data, however, as the size of the data exported in a single export is very large, the bulk of the reasoning process time is spent on exporting the data. Moreover, the evaluation did not notice any improvements between using 8 cores or 16 cores VMs, however an 8 cores VM provides better results than a 4 cores VM.

7.6.4 BigQuery Data Import and Export

Importing and exporting data into and from BigQuery are free operations that run asynchronously without any guarantee on how long the operations take. Figure 7.6 shows the BigQuery import and export times versus the number of retrieved and inserted rows respectively for both the DBpedia and LUBM datasets. As seen, when exporting a large number of rows in one operation, the export can take variable time ranging from 600 seconds to 2000 seconds. Moreover, importing a small number of rows into BigQuery can occasionally take a proportionally long time compared with the normal export times. For example, as shown in Figure 7.6(b) this can take more

than 600 seconds to complete, whereas the majority of the import for smaller rows is completed within 200 seconds.

7.6.5 Comparison of Forward RDFS Reasoning Throughput

In terms of reasoning throughput, both Tables 7.10 and 7.11 report on the processor reasoning throughput and the overall reasoning throughput for the LUBM and DBpedia datasets respectively. The processor reason throughput reports the time required by the processor exclusively to infer new triples. The end to end reason throughput is based on the total runtime for the `DoReasonTask` as reported by the coordinator, which includes a number of inference cycles, the time required to run queries and to import and export the BigQuery results. To return to the subject of the processor reason time, it can be seen that the reported throughput is very hight compared to the end to end throughput, primarily due to the fact that the remainder of the time is spent in importing and exporting data into and from BigQuery.

Table 7.13 provides comparison of the reasoning throughput reported by other large scale distributed reasoning based on dedicated hardware. As can be seen for the LUBM dataset the approach proposed in this thesis provides a throughput of 1,024 which in slightly below the results reported by others. However, these approaches [128, 139, 21] utilise a file based approach, where the inferred files are yet to be imported in a triple store so that the data can be queried. Kotoulas, et al. [19] on the other hand, utilise a memory based approach without any indication on long term storage or query support for the inferred data. Although, the approach developed in this thesis takes slightly longer to complete the reasoning process, the trade-off is that the inferred data is uploaded to BigQuery and can immediately be queried, hence enabling a full triple store capability. In contrast to [19, 128, 139, 21], where a higher throughput is reported, but the data is yet to be imported into a system that

Table 7.13: Comparison of Forward RDFS Reasoning

Work	Dataset	Throughput Ktps	Setup
Weaver. J [21]	LUBM	1,185	128 processors on supercomputer
Urbani. J [128]	LUBM	1,931	32 nodes Hadoop cluster
This Approach	LUBM	1,024	16 cloud VMs
Kotoulas et al. [19]	Real-world	450	64 nodes cluster
Urbani et al. [139]	Real-world	252	32 nodes Hadoop cluster
This Approach	DBpedia	255	16 cloud VMs

enables the data to be queried.

Additionally, the approach presented in this thesis also outperforms DHT based approaches, for example, [16] approach was not able to deal with more than 10K triples in forward reasoning mode, whilst [15] reports the closure of only 2M triples in 28 minutes on a 32 nodes system. To conclude, the forward reasoning approach developed in this thesis provides a promising mechanism for utilising cloud virtual machines and columnar databases like BigQuery to not only carryout forward reasoning, but to also provide storage and triple store capabilities.

7.6.6 Forward Reasoning Conclusion

To summarise, the BigQuery forward reasoning approach developed in this thesis is viable and provides promising results when compared with other related work. However, this approach is impacted by the BigQuery variable import and export times. Having said this, at the time of writing Google has announced a new BigQuery mechanism called *Poseidon*[6], which is claimed to improve export and import times by 5x, this can considerably improve the forward reasoning.

Additionally, load balancing the reasoning process using the selective term strategy needs fine tuning to cater for reasoning cycles 2 to n. Moreover, the inferred duplicates

[6] `https://cloudplatform.googleblog.com/2016/03/Google-takes-Cloud-Machine-Learning-service-mainstream.html`

triples do also impact the reasoning process and generate unnecessary traffic between the processors and BigQuery, hence addressing this issue will improve the reasoning process considerably. Finally, it was observed that the data export and import from BigQuery in the form of files downloaded and uploaded to Cloud Storage is currently done sequentially rather in parallel. Downloading and uploading files in parallel from and to Cloud Storage during the reasoning process will save on average 5 minutes for both the DBpedia and LUBM datasets.

7.7 CloudEx Evaluation

In the context of this thesis the CloudEx framework is evaluated by utilising it for the processing of RDF, including dictionary encoding, loading and forward reasoning. As noted in Chapter 3, cloud computing charges consumer for the resources they use, in this regards the Google Cloud Platform enables users to export detailed usage reports of the various cloud services. Using these reports it was found that over the duration of this research the CloudEx framework has acquired 1,086 virtual machines[7] for RDF processing. With the availability of detailed billing reports for the various cloud services used, it was also possible to calculate the overall cost of the experimentation phase, which was found to be exactly $290.29.

7.7.1 CloudEx Improvements

The CloudEx framework successfully acquired and sent instructions to processors using the Google Compute Engine APIs, the number of acquired processors ranges from 1 to 16 with various memory and CPU cores configurations. As was noted in Section 7.4.2 the metadata mechanism used by CloudEx requires on average 20

[7] `http://omerio.com/2016/03/16/saving-hundreds-of-hours-with-google-compute-engine-per-minute-billing/`

seconds to send instructions to processors due to the time required to invoke the Compute Engine metadata API. However, as CloudEx invokes this APIs serially, the latencies of sending instructions to the processors increases with increasing the number of processors. Consequently, adding improvements to invoke the metadata API in parallel for all the processors will ensure that the time required to send instructions to the processors remains constant regardless of the number of processors. Similarly, as was noted in Section 7.6.6, the time required to download and upload files to Cloud Storage can be improved considerably by processing these files in parallel rather than serially.

7.7.2 Cloud Platform Observations

It was found that the average *acquisition time* t_{aq} taken to acquire CloudEx processors is 85 seconds. This is the time taken for the processor to be created, started and runs the CloudEx application. Additionally, the average network bandwidth between the processors and Cloud Storage was measured when downloading the DBPedia dataset files from Cloud Storage to the processors VMs. The average observed bandwidth was 375 Mbit/s and the maximum observed bandwidth was 554 Mbit/s. It was also noted that files larger than 150 MB achieve a better bandwidth than smaller files.

7.7.3 Cost Considerations

One of the additional considerations when utilising cloud computing is cost, therefore, linear speedup enables CloudEx to utilise more processors to reduce the runtime but at the same time keep the cost fixed. For example, if the hourly cost of a VM is c and a task takes t hours to execute on one VM, the total cost is $c \times t$. From Formula 7.2, a linear speedup means that the task execution time on n VMs is $\frac{t}{n}$, consequently the total cost in this case is $c \times n \times \frac{t}{n} = c \times t$. The cost is the same as when using one

VM, but the task is done n times quicker. Consequently, the primary focus of the RDF distributed processing employed in this chapter was on optimising the various tasks to provide linear or close to linear scalability.

In addition to the VM hourly pricing, cloud providers specify other cost items related to storage, data transfers, etc., but in terms of CloudEx processing, the bulk of the cost is usually related to the VMs and BigQuery. Other costs such as storage and network transfers are negligible in comparison. As it was noted in Section 3.4.1.2, the Google Cloud Platform charges for VMs per minute with a 10 minutes minimum charge. This means if the overall runtime of the CloudEx processors is less than 10 minutes, even if the coordinator shuts them down before the end of the 10 minutes there will still be a charge for 10 minutes. This will be exacerbated with other cloud providers that charge for VMs per hour basis rather than per minute, in which case there need to be the ability to only shutdown CloudEx processors after a full hour of operation. Using the usage reports for the experimentation work done in this chapter, it was found that using CloudEx with a cloud provider that charges per hour — such as Amazon Web Services — will result in additional 697 virtual machine hours being billed. So per-minute billing for cloud virtual machines have saved this research 697 hours[8].

7.8 Summary

This chapter provided experimental evaluation of the ECARF algorithms outlined in Chapter 6, which uses the CloudEx framework (presented in Chapter 5) to execute tasks and jobs on the Google Cloud Platform. The evaluation presented in this chapter has utilised both large and small, real-world and synthetic datasets to evaluate the

[8] `http://omerio.com/2016/03/16/saving-hundreds-of-hours-with-google-compute-engine-per-minute-billing/#Appreciating_the_Saved_Hours`

contributions of this thesis. The evaluation also reported on a number of optimisations that have been made to improve the algorithms developed in this thesis. Where possible, comparison with other work was also provided to highlight the feasibility and effectiveness of these algorithms. The evaluation has focused on the following four areas shown with their relevant research questions (summarised in Section 7.1) in order to evaluate the contributions of this thesis:

- Distributed RDF processing (Q1, Q2), presented in Section 7.4.
- Dictionary encoding (Q2), presented in Section 7.5.
- Forward reasoning (Q3), presented in Section 7.6.
- CloudEx evaluation (Q1), presented in Section 7.7.

In this chapter it was shown that the algorithms and designs that form the contributions of this thesis are feasible and effective and in many instances, with only 16 cloud virtual machines, outperform other related approaches that are based on large clusters based on dedicated hardware. The following chapter concludes this thesis, summarises the work that has been done and suggests further work.

Chapter 8

Conclusions

The Semantic Web which was proposed as an extension to the traditional Web, promised to revolutionise the way applications deal with Web data. With the Semantic Web applications can be more intelligent and relevant to user requirements by providing relevant contents based on context. Not only this, but such applications can also provide intelligent answers based on the meaning of data, not just based on textual matches. However, the explosive growth of data on the Web has resulted in a similar growth in the Semantic Web RDF data, to the extent that typical applications now need to deal with billions of RDF statements. Existing approaches have focused on distributed processing using dedicated hardware and computer clusters. These approaches need to constantly deal with the increasing size of RDF data by acquiring more storage and computing hardware. Having said this, little research has focused on the cloud computing paradigm.

This thesis has utilised cloud computing to address some of the issues facing large scale RDF processing. The aim of this research is to address the constant growth of this RDF data by developing a solution that is able to scale based on cloud computing resources. More specifically the aim is to develop and evaluate an elastic cloud-

based triple store for RDF processing in general and RDFS rule-based reasoning in particular. As a result the following research questions were formulated:

- Q1. How can a cloud-based system efficiently distribute and process tasks with different computational requirements?
- Q2. How can an efficient dictionary that fits in-memory be created for encoding RDF URIs from strings to integers?
- Q3. How can cloud-based big data services such as Google BigQuery be used for RDFS rule-based reasoning?

This thesis has answered these questions by providing a number of contributions which can be summarised as the design and implementation of the **Cloud**-based Task **Ex**ecution framework (CloudEx) and the **E**lastic **C**ost **A**ware **R**easoning **F**ramework (ECARF) triple store. CloudEx, is a cloud-first framework for tasks execution using cloud computing services, on the other hand ECARF is a cloud-based triple store for RDF processing and rule-based reasoning. The thesis has also introduced the notion *cloud-first frameworks* to denote frameworks that are entirely cloud based and utilise cloud features, such as elasticity, and services, such as virtual machines, cloud storage and big data services. Unlike physical computing, when considering cloud computing users can acquire any number of computational resource — subject to the cloud provider usage limits — but at a cost. Henceforth, the main requirements for utilising cloud computing is to optimise resource usage to reduce both the cost and runtime.

This final chapter concludes the thesis by presenting a summary of the contributions both in the research and practical fields. Additionally, the chapter discusses open issues and outlines possible future work. The following sections summarise each of

the research questions and outline the contributions developed as part of this thesis in order to answer these questions.

8.1 The CloudEx Framework Contributions

This section summarises the contributions related to the CloudEx framework that provided the answer to this research question:

- Q1. How can a cloud-based system efficiently distribute and process tasks with different computational requirements?

8.1.1 Summary of CloudEx

CloudEx is a generic cloud based task execution framework that can be implemented on any public cloud provider that offers virtual machines service. CloudEx enables users to use a divide-conquer approach to break down their jobs to a sequence of tasks, some to be run in serial and others in parallel. For the later users specify a partitioning function to be used to distribute the task execution between a number of ephemeral virtual machines. Not only this, but users can also define the computational requirements for these parallel tasks in terms of CPU cores and memory. CloudEx provides a workload partitioning algorithm based on a variation of the Bin Packing algorithm [124, 125] that users can use to distribute the processing of tasks. CloudEx enables users to dynamically adjust virtual machine specification and partitioning functions for upcoming tasks in a job based on the outcome of previous tasks by using feedback. The key points to note about CloudEx:

- It is a cloud-first framework which is entirely based on cloud computing features, such that resources can be acquired, released and manipulated programmatically using an API.

- Central orchestration using a *coordinator* is used to coordinate between the various computing resources, manage workload distribution and to release resources when not needed.

- Computing nodes started by the *coordinator* — called *processors* — are only used for processing and treated as ephemeral, such that once the processing is done these resources are shutdown. Additionally these *processors* do not store any data locally and do not exchange data with each other by following an "embarrassingly parallel" approach.

8.1.2 Effectiveness and Feasibility

In this thesis the CloudEx framework was evaluated in the context of the distributed processing of RDF and proven to be viable, demonstrating its ability to acquire from 1 to 16 processors with various memory and CPU configurations. Additionally, it was shown that the CloudEx bin packing partitioning algorithm provides a flexible mechanism for distributing task execution between multiple processors. More importantly, over the duration of the evaluation phase, the CloudEx framework was able to acquire 1,086 virtual machines to process RDF at an overall cost of $290.29. Proving that CloudEx provides a viable and efficient mechanism to acquire cloud based computational resource when needed and released once the processing is done to avoid incurring unnecessary cost.

8.2 The ECARF Triple Store Contributions

This section summarises the contributions related to the ECARF triple store that provided the answers to these research questions:

- Q2. How can an efficient dictionary that fits in-memory be created for encoding RDF URIs from strings to integers?

- Q3. How can cloud-based big data services such as Google BigQuery be used to perform RDFS rule-based reasoning?

The ECARF distributed RDF processing algorithms were evaluated in this thesis, with particular focus on the partitioning factor for each of the algorithms when processing real-world datasets. In this regards, it was shown that the load algorithms exhibit linear scalability for the LUBM dataset and outperforms other reported results on dedicated hardware. This is primarily due to a combination of using the CloudEx's workload partitioning algorithm, its ability to acquire VMs with various configurations and the efficient dictionary encoding summarised below.

8.2.1 Summary of Dictionary Encoding

This thesis has designed and implemented a cloud-based algorithm for generating an efficient dictionary for RDF URIs compression and decompression using dictionary encoding to encode values from strings to integers. The redundancy in the generated dictionary is removed by ensuring that common parts of the RDF URIs are encoded separately. The two URI parts are then joined into one number by using a "bitwise" pairing function [131]. The optimal size of the dictionary helps to hold it in memory by utilising high memory cloud virtual machines, hence speeding up the dataset encoding process considerably. Although dictionary encoding is a common place in Semantic Web reasoners the approach described here is entirely cloud-based and focuses on the size of generated dictionary by using a pairing function to encode URI parts.

8.2.2 Effectiveness of Dictionary Encoding

The ECARF dictionary encoding was shown to outperform other reported work, both in terms of the size of the produced dictionary and the overall compression rate of the encoded datasets. The ECARF dictionary encoding algorithm was shown to be the primary factor behind the ECARF efficient RDF datasets load due the efficient size of the dictionary produced. This facilitates the in-memory handling of the dictionary and hence provide better performance than in-file based approaches.

8.2.3 Summary of Forward Reasoning

This thesis has presented a number of algorithms for RDF processing and RDFS rule-based reasoning using a number of computing nodes and columnar databases [88], in particular Google BigQuery [31]. This approach distributes the reasoning process using a selective term strategy, then each node retrieves the data required to generate the inferred statements using SQL like queries, generates the inferred data and updates the columnar storage accordingly. In the context of distributed RDFS rule-based reasoning, this thesis defined a selective term partitioning strategy for partitioning the reasoning workload between a number of processors. This strategy builds on the partitioning strategies developed for DHTs based algorithms [55] and the following:

- The use of the Bin Packing algorithm to ensure that all processors have an approximately equal workload.
- The ability to pre-process and analyse the RDF data upfront enables the gather of details that facilitates strategy such the productive instance triples count.
- The removal of the data storage from the processors to cloud-based storage and

columnar databases alleviate the processors from the storage challenges as a result of the data skew.

8.2.4 Effectiveness of Forward Reasoning

Based on the evaluation presented in this thesis, it was shown that the ECARF forward reasoning algorithm based on Google BigQuery provide a viable approach when compared with other related work. The algorithm provides the benefit of permanent storage for RDF datasets with the potential to query and develop a full triple store capability on top of BigQuery. It was shown that the reasoning algorithm provides a very high reasoning throughput per processor, but with a reduced overall throughput when considering the time taken by BigQuery. This is primarily due to the import and export time taken by BigQuery and the fact that the selective term partitioning strategy does not provide an optimal partitioning approach after the first reasoning cycle. However, this reduced end to end throughput provides the benefit of permanent storage for the dataset with the potential to query and develop a full triple store capability on top of BigQuery. In comparison, other related work report a slightly higher reasoning throughput on approaches that are either in-memory or file based, with the data yet to be imported into a permanent storage with triple store capability.

8.3 Open Source Contributions and Impact

The CloudEx framework architecture and algorithms are implemented as a publicly available open source framework[1]. Additionally, ECARF is implemented on the Google Cloud Platform as a publicly available open source framework[2]. ECARF is the first to run RDFS reasoning entirely on cloud computing, unlike other approaches

[1] `https://cloudex.io/`
[2] `http://ecarf.io`

that simply replicate physical computing setup — such as MapReduce [20] — on the cloud, ECARF is entirely based on cloud computing features and services and can immediately be utilised by any applications that require triple store capabilities without any upfront investment in hardware or software licences. Some of the findings of this research have already attracted a great deal of interest (more than 10,000 views over a few days) from the technical community, in particular to potential savings with public clouds that charge per-minutes versus the ones that charge per-hour[3].

8.4 Future work

This section outlines possible future directions for the work carried out in this research alongside any open issues with the approaches presented.

8.4.1 CloudEx Improvements

A number of CloudEx improvements have been summarised in Section 7.7.1, mainly so that the coordinator can issue metadata updates in parallel rather than serially. This approach ensures that the VM acquisition time is constant regardless of the number of processors. Additionally, it was shown that the CloudEx implementation for the Google Cloud Platform can benefit from implementing parallel handling for uploading and downloading files to and from Cloud Storage. Furthermore, this thesis has evaluated CloudEx in the context of processing RDF datasets, yet, future work can evaluate CloudEx in other contexts and potentially provide comparison with other frameworks such a MapReduce. The following sections suggest further improvements to ensure CloudEx is a production ready and resilient framework.

[3] `http://omerio.com/2016/03/16/saving-hundreds-of-hours-with-google-compute-engine-per-minute-billing/`

8.4.1.1 Resilience

The current implementation of CloudEx does not provide any coordinator resilience, future work is needed to provide resilience for the coordinator and job persistence in a database. In this case, a backup coordinator can be used to carry on with the job execution if the main coordinator fails. There is also a need for retrying failed processors rather than terminating the whole job with an error status.

8.4.1.2 Autoscaling of Resources

One of the key improvements to consider is the ability to autoscale the processors vertically or horizontally based on time deadlines and budget constraints when distributing the workload between processors. This can involve the acquisition of a variable number of virtual machine with either lower or higher specification, or a mix, to meet particular time deadlines or budget constraints. In this regards, using CloudEx with cloud providers that charge per hour for virtual machines presents its own challenges because Google Cloud Platform charges for resources per minute. Subsequently, implementation for other providers needs to cache processors for a full hour before shutting them down.

8.4.1.3 Implementation for Other Clouds

Another improvement is to provide implementation for other prominent cloud providers such as Amazon Web Services and Microsoft Azure. One of the challenges of using cloud computing is that internal implementation varies between one cloud provider to another. Having said this, the results reported in this thesis are specific to the Google Cloud Platform and it will be interesting, once the CloudEx implementation for other platforms is developed, to run experiments on other public clouds and compare their results. Another avenue for future work is to utilise Container technology,

commonly known as Docker[4], in creating CloudEx processors rather than virtual machines. Containers utilise operating system level virtualisation and have the potential of improved performance compared to full virtualisation as in virtual machine. This is specially true if containers are run directly on top of the physical host's operating system.

8.4.2 ECARF Improvements

A number of improvements have been suggested in Chapter 7 in relation to the algorithms implemented as part of ECARF, namely the dictionary encoding and forward reasoning using Google BigQuery. The dictionary encoding improvements include the ability to automatically acquire a VM with sufficient memory to assemble the dictionary. Moreover, in terms of the forward reasoning algorithm, the ability to download and upload the BigQuery data files in parallel will speed up the reasoning process by up to 4 minutes. Furthermore, the relevant term partition strategy for the reasoning process requires further improvements to ensure it can provide load balancing on all reasoning cycles, and not just for the first one. The following sections suggest further improvements to ECARF to ensure it can provide full triple store capability.

8.4.2.1 Triple Store Capability

The rule-based reasoning supported by ECARF can be extended to cover richer semantics such as the OWL Horst [48] and OWL 2 [52] rulesets. Moreover, additional activities such data management and query support can be added to ECARF to provide full triple store capability. Data management provide the ability to create, update and delete statements on the dataset, on the other hand, query support provides SPARQL [11] support on top of Google BigQuery. Furthermore, the dictionary

[4] `https://www.docker.com/`

encoding algorithm can further remove any redundancy in the dictionary by analysing the frequency of certain URI sections. Then, instead of always splitting these URIRef at the last slash, they could be split at a variable location to minimise any redundancy.

8.4.2.2 Redundant Triples

The approach presented in this thesis for forward reasoning does not eliminate redundant *rdf:type* instance triples, which are mainly generated due to many of the *rdfs:domain* and *rdfs:subClassOf* schema triples sharing the same object. For example, applying the *SI* rules in Table 6.4 to the following instance triples: *T(s1 p1 o1)*, *T(s1 p2 o2)* and schema triples: *T(p1 rdfs:domain o3)*, *T(p2 rdfs:domain o3)* respectively, will generate two duplicate *T(s1 rdf:type o3)* triples.

Several approaches have been proposed to either eliminate duplicate triples after they are generated [19], or to avoid generating them in the first place [20]. Since in this thesis processors do not communicate with each other, an effective approach is to avoid generating redundant triples. By grouping together the *rdfs:domain* and *rdfs:subClassOf* schema triples that share the same object, one query can be issued that has an *OR* condition in the *where* clause and when the results are retrieved the algorithm simply does not generate duplicate triples. For the example triples presented earlier, such query will be in the form "*SELECT subject FROM Triple WHERE predicate = p1 or p2*". This approach will also require the schema triples that share the same object to be assigned to the same processor, so the impact of this on the selective term partitioning outlined in this thesis will need to be assessed as well.

References

[1] Vijay K. Vaishnavi and William Kuechler, Jr. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Auerbach Publications, Boston, MA, USA, 1st edition, 2007.

[2] Tim Berners-Lee, J Hendler, and O Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[3] G. Klyne, J. J. Carroll, and B. McBride. Resource Description Framework (RDF): Concepts and Abstract Syntax. *W3C Recommendation*, 2004.

[4] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. pages 1–15, 2004.

[5] Dave Beckett and Brian McBride. RDF/XML Syntax Specification (Revised). *W3C Working Draft*, (October):1–56, 2003.

[6] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax, 2011.

[7] Patrick Hayes. RDF Semantics. *W3C Recommendation*, 10:1–45, 2004.

[8] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef,

Sören Auer, and Christian Bizer. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[9] Jem Rayfield. BBC World Cup 2010 dynamic semantic publishing. `http://www.bbc.co.uk/blogs/bbcinternet/2010/07/bbc_world_cup_2010_dynamic_sem.html`, 2010. [Online; accessed 13-December-2015].

[10] Jem Rayfield. Sports Refresh: Dynamic Semantic Publishing. `http://www.bbc.co.uk/blogs/bbcinternet/2012/04/sports_dynamic_semantic.html`, 2012. [Online; accessed 13-December-2015].

[11] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. *W3C Recommendation*, 2009(January):1–106, 2008.

[12] Sam Madden. From databases to big data. *Internet Computing, IEEE*, 16(3):4–6, May 2012.

[13] The Semantic Web Challenge. `http://challenge.semanticweb.org/`, 2015. [Online; accessed 13-December-2015].

[14] LargeTripleStores - W3C Wiki. `http://www.w3.org/wiki/LargeTripleStores`. [Online; accessed 13-December-2015].

[15] Qiming Fang, Ying Zhao, Guangwen Yang, and Weimin Zheng. Scalable distributed ontology reasoning using DHT-based partitioning. *The Semantic Web*, pages 91–105, 2008.

[16] Zoi Kaoudi and Manolis Koubarakis. Distributed RDFS Reasoning Over Structured Overlay Networks. *Journal on Data Semantics*, 2(4):189–227, mar 2013.

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] Aidan Hogan, Jeff Z Pan, Axel Polleres, and Stefan Decker. SAOR : Template Rule Optimisations for Distributed Reasoning over 1 Billion Linked Data Triples ? *The Semantic Web–ISWC 2010*, 1380:337–353, 2010.

[19] Spyros Kotoulas, Eyal Oren, and Frank Van Harmelen. Mind the data skew: distributed inferencing by speeddating in elastic regions. *Proceedings of the 19th international conference on World wide web*, pages 531–540, 2010.

[20] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank Van Harmelen, and Henri Bal. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *Web Semantics: Science, Services and Agents on the World Wide Web*, 10:59–75, jan 2012.

[21] Jesse Weaver and J. Hendler. Parallel materialization of the finite rdfs closure for hundreds of millions of triples. *The Semantic Web-ISWC 2009*, pages 682–697, 2009.

[22] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43, feb 2003.

[23] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing, Recommendations of the National Institute of Standards and Technolog. *National Institute of Standards and Technology*, 2011.

[24] M Armbrust, A Fox, R Griffith, AD Joseph, and RH. Above the clouds: A Berkeley view of cloud computing. *University of California, Berkeley, Tech. Rep. UCB*, pages 07–013, 2009.

[25] L Schubert, K Jeffery, and B Neidecker-Lutz. The Future of Cloud Computing. Opportunities for European Cloud Computing Beyond 2010. *European Commission, the Cloud Expert Group*, page 66, 2010.

[26] Jinesh Varia. Architecting for the cloud: Best practices. *Amazon Web Services*, (May):1–21, 2010.

[27] Inc. Amazon Web Services. Amazon web services (aws) - cloud computing services. `https://aws.amazon.com/`, 2015. [Online; accessed 13-December-2015].

[28] Google Developers. Google cloud computing, hosting services & apis. `https://cloud.google.com/`, 2015. [Online; accessed 13-December-2015].

[29] Azure.microsoft.com. Microsoft azure: Cloud computing platform & services. `https://azure.microsoft.com/`, 2015. [Online; accessed 13-December-2015].

[30] Inc. Amazon Web Services. Aws — amazon redshift - cloud data warehouse solutions. `http://aws.amazon.com/redshift/`, 2015. [Online; accessed 13-December-2015].

[31] Google Developers. Bigquery - large-scale data analytics. `https://cloud.google.com/products/bigquery/`, 2015. [Online; accessed 13-December-2015].

[32] Sergey Melnik, Andrey Gubarev, and JJ Long. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[33] Paul Marshall, HM Tufo, and Kate Keahey. Architecting a Large-scale Elastic Environment-Recontextualization and Adaptive Cloud Services for Scientific Computing. *ICSOFT*, 2012.

[34] Gavin Carothers. RDF 1.1 N-Quads - A line-based syntax for an RDF datasets. 2014.

[35] Jan Grant and Dave Beckett. RDF Test Cases. *W3C Recommendation*, 2004. Available at `http://www.w3.org/TR/2004/REC-rdf-testcases-20040210`.

[36] Pascal Hitzler, Markus Krtzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 1st edition, 2009.

[37] Frank van Harmelen Deborah L. McGuinness. Owl web ontology language overview. *W3C recommendation 10.2004-03*, 2004(February):1–12, 2004.

[38] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[39] Nicola Guarino and Pierdaniele Giaretta. Ontologies and Knowledge Bases. *Towards Very Large Knowledge Bases: Knowledge Building & Knowledge Sharing*, pages 25–32, 1995.

[40] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics as Ontology Languages for the Semantic Web. *Mechanizing Mathematical Reasoning*, pages 228–248, 2005.

[41] Nicola Guarino. Formal Ontology and Information Systems. *Proceedings of the first international conference*, 46(June):3–15, 1998.

[42] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5-6):907–928, 1995.

[43] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*, volume 32. 2010.

[44] Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 2000:1–34, 2001.

[45] Markus Krötzsch, F Simancik, and Ian Horrocks. Description Logic Primer. *arXiv preprint arXiv:1201.4089*, (January):1–16, 2012.

[46] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The Even More Irresistible SROIQ. *Proc. 10th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 57–67, 2006.

[47] Michael Sipser. *Introduction to the Theory of Computation*, volume 27. 1997.

[48] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):79–115, oct 2005.

[49] BN Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: Combining logic programs with description logic. *Proceedings of the 12th international conference on World Wide Web*, pages 48–57, 2003.

[50] B Motik, BC Grau, I Horrocks, and Z Wu. Owl 2 web ontology language: Profiles. *W3C . . .*, 2009.

[51] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL Envelope. *IJCAI*, jan 2005.

[52] M Krötzsch. OWL 2 Profiles: An introduction to lightweight ontology languages. *Reasoning Web. Semantic Technologies for Advanced Query Answering*, 7487:112–183, 2012.

[53] J Hendler. On beyond ontology, in keynote talk, second international semantic web conference. `http://www.cs.rpi.edu/~hendler/LittleSemanticsWeb.html`, 2003. [Online; accessed 13-December-2015].

[54] S Muñoz, J Pérez, and C Gutierrez. Simple and Efficient Minimal RDFS. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220–234, 2009.

[55] Tugba Kulahcioglu and Hasan Bulut. Overcoming limitations of term-based partitioning for distributed RDFS reasoning. In *Proceedings of the Fifth Workshop on Semantic Web Information Management - SWIM '13*, SWIM '13, pages 1–4, New York, New York, USA, 2013. ACM Press.

[56] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, and Ivan Peikov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

[57] Orri Erling. Virtuoso , a Hybrid RDBMS / Graph Column Store. *Bulletion of the IEEE Society Technical Committee on Data Engineering*, pages 1–6, 2013.

[58] Vladimir Kolovski, Zhe Wu, and George Eadon. Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. *The Semantic Web–ISWC 2010*, 2010.

[59] Min Cai and Martin Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. *Proceedings of the 13th international conference on ...*, pages 650–657, 2004.

[60] D Battré, F Heine, A Höing, and O Kao. On triple dissemination, forward-chaining, and load balancing in DHT based RDF stores. *... Information Systems, and Peer-to-Peer ...*, 2007.

[61] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS reasoning and query answering on top of DHTs. *The Semantic Web-ISWC 2008*, 2008.

[62] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Frank Van Harmelen, Annette ten Teije, and Frank van Harmelen. Marvin: Distributed reasoning over large-scale Semantic Web data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(4):305–316, dec 2009.

[63] Tugba Kulahcioglu and Hasan Bulut. On scalable RDFS reasoning using a hybrid approach. *Turkish Journal of Electrical Engineering & Computer Sciences*, pages 1–15, 2016.

[64] Jacopo Urbani. Scalable and parallel reasoning in the Semantic Web. *The Semantic Web: Research and Applications*, pages 488–492, 2010.

[65] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, F. Van Harmelen, and H. Bal. OWL reasoning with WebPIE: calculating the closure of 100 billion triples. *The Semantic Web: Research and Applications*, pages 213–227, 2010.

[66] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, (610):1–34, 2005.

[67] Jacopo Urbani and Jason Maassen. Scalable RDF data compression with MapReduce. *Concurrency and Computation: Practice and Experience*, (April 2012):24–39, 2013.

[68] Batselem Jagvaral and Young Tack Park. Distributed scalable RDFS reasoning. *2015 International Conference on Big Data and Smart Computing, BIGCOMP 2015*, pages 31–34, 2015.

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[70] Aidan Hogan, Andreas Harth, and Axel Polleres. Scalable Authoritative OWL Reasoning for the Web. *International Journal on Semantic Web and Information Systems*, 5(2), 2009.

[71] Sören Auer, Jens Lehmann, Axel Cyrille Ngonga Ngomo, and Amrapali Zaveri. Introduction to linked data and its lifecycle on the web. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8067 LNAI:1–90, 2013.

[72] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Elsevier/Morgan Kaufmann, 2008.

[73] Gregory F Pfister. An Introduction to the InfiniBand Architecture. *High Performance Mass Storage and Parallel {I/O}: Technologies and Applications*, (42):617–632, 2001.

[74] A Aranda-Andújar. AMADA: web data repositories in the amazon cloud. *CIKM '12 Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 3–5, 2012.

[75] Zoi Kaoudi and Ioana Manolescu. Triples in the clouds. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1258–1261, apr 2013.

[76] Francesca Bugiotti, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. RDF data management in the Amazon cloud. *Proceedings of the 2012 Joint EDBT/ICDT Workshops on - EDBT-ICDT '12*, page 61, 2012.

[77] Inc. Amazon Web Services. Aws — amazon simpledb - simple database service. `https://aws.amazon.com/simpledb/`, 2015. [Online; accessed 13-December-2015].

[78] Raffael Stein and Valentin Zacharias. Rdf on cloud number nine. *4th Workshop on New Forms of Reasoning ... ...*, pages 1–13, 2010.

[79] Kyriakos Kritikos, Yannis Rousakis, and Dimitris Kotzinos. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XX: Special Issue on Advanced Techniques for Big Data Management*, chapter A Cloud-Based, Geospatial Linked Data Management System, pages 59–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[80] M Oberhumer. LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo/`, 2016. [Online; accessed 18-June-2016].

[81] Jesse Weaver. *Toward Webscale, Rule-Based Inference On The Semantic Web Via Data Parallelism*. PhD thesis, Rensselaer Polytechnic Institute, February 2013.

[82] Zoi Kaoudi, Kostis Kyzirakos, and Manolis Koubarakis. SPARQL query optimization on top of DHTs. *The Semantic Web–ISWC 2010*, 2010.

[83] JD Fernández. RDF compression: basic approaches. *Proceedings of the 19th international conference on World wide web*, pages 3–4, 2010.

[84] Kisung Lee, Jin Hyun Son, Gun-Woo Kim, and Myung Ho Kim. Web document compaction by compressing URI references in RDF and OWL data. *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pages 163–168, 2008.

[85] Jacopo Urbani. Three Laws Learned from Web-scale Reasoning. *2013 AAAI Fall Symposium Series*, pages 76–79, 2013.

[86] Xenproject.org. Xen project. `http://www.xenproject.org/`, 2015. [Online; accessed 13-December-2015].

[87] Linux-kvm.org. Kvm. `http://www.linux-kvm.org/`, 2015. [Online; accessed 13-December-2015].

[88] DJ Abadi, PA Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[89] Inc. Amazon Web Services. Elastic compute cloud (ec2) cloud server & hosting. `http://aws.amazon.com/ec2/`, 2015. [Online; accessed 13-December-2015].

[90] Google Developers. Compute engine - iaas. `https://cloud.google.com/products/compute-engine/`, 2015. [Online; accessed 13-December-2015].

[91] Azure.microsoft.com. Virtual machines linux & windows vms — microsoft azure. `https://azure.microsoft.com/en-gb/services/virtual-machines/`, 2015. [Online; accessed 13-December-2015].

[92] Inc. Amazon Web Services. Amazon simple storage service (s3) - object storage. `http://aws.amazon.com/s3/`, 2015. [Online; accessed 13-December-2015].

[93] Google Developers. Cloud storage - online data storage. https://cloud.google.com/products/cloud-storage/, 2015. [Online; accessed 13-December-2015].

[94] Adam Jacobs. The Pathologies of Big Data. *Queue*, 7(6):10, July 2009.

[95] Gartner Inc. Gartner's 2013 hype cycle for emerging technologies maps out evolving relationship between humans and machines. http://www.gartner.com/newsroom/id/2575515, August 2013. [Online; accessed 13-December-2015].

[96] Google Developers. Machine Types - Compute Engine – Google Cloud Platform. https://cloud.google.com/compute/docs/machine-types, 2015. [Online; accessed 13-December-2015].

[97] Google Developers. Storing and retrieving instance metadata. https://cloud.google.com/compute/docs/metadata#waitforchange, 2015. [Online; accessed 13-December-2015].

[98] Google Developers. Cloud storage pricing. https://cloud.google.com/storage/pricing, 2015. [Online; accessed 13-December-2015].

[99] Kazunori Sato. An Inside Look at Google BigQuery. Technical report, 2012.

[100] Google Developers. Query reference. https://cloud.google.com/bigquery/query-reference, 2015. [Online; accessed 13-December-2015].

[101] Google Developers. Table decorators. https://cloud.google.com/bigquery/table-decorators, 2015. [Online; accessed 13-December-2015].

[102] Google Developers. Bigquery pricing. https://cloud.google.com/bigquery/pricing#data, 2015. [Online; accessed 13-December-2015].

[103] Christina Hoffa, Gaurang Mehta, Timothy Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. In *Proc. 4th IEEE Int. Conf. on eScience (eScience - 2008)*, pages 640–645, 2008.

[104] Viktor Mauch, Marcel Kunze, and Marius Hillenbrand. High performance cloud computing. *Future Generation Computer Systems*, 29(6):1408–1416, aug 2013.

[105] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. *Computing*, 22(December):115–131, 2010.

[106] Carlos De Alfonso, Miguel Caballer, Fernando Alvarruiz, Germ X B N Molto, and Vicente Hern X B Ndez. Infrastructure Deployment Over the Cloud. *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 517–521, 2011.

[107] Miguel Caballer, Carlos de Alfonso, Fernando Alvarruiz, and Germán Moltó. EC3: Elastic Cloud Computing Cluster. *Journal of Computer and System Sciences*, 79(8):1341–1351, dec 2013.

[108] Miguel Caballer, Ignacio Blanquer, Germán Moltó, and Carlos de Alfonso. Dynamic Management of Virtual Infrastructures. *Journal of Grid Computing*, pages 53–70, 2014.

[109] Anastasia Grekioti and NataliaV. Shakhlevich. Scheduling bag-of-tasks applications to optimize computation time and cost. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski, editors, *Parallel Processing and Applied Mathematics*, volume 8385 of *Lecture Notes in Computer Science*, pages 3–12. Springer Berlin Heidelberg, 2014.

[110] Ana Maria Oprescu and Thilo Kielmann. Bag-of-tasks scheduling under budget constraints. *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010*, pages 351–359, 2010.

[111] M Mao, J Li, and Marty Humphrey. Cloud Auto-scaling with Deadline and Budget Constraints. *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference*, pages 41–48, 2010.

[112] Google Developers. Cloud dataflow - google cloud platform. `https://cloud.google.com/dataflow/`, 2015. [Online; accessed 13-December-2015].

[113] Inc. Amazon Web Services. Aws data pipeline - amazon web services. `https://aws.amazon.com/datapipeline/`, 2015. [Online; accessed 13-December-2015].

[114] Inc. Amazon Web Services. Aws — high performance computing - hpc cloud computing. `http://aws.amazon.com/hpc/`, 2015. [Online; accessed 13-December-2015].

[115] Azure.microsoft.com. Microsoft azure — big compute: Hpc and batch. `https://azure.microsoft.com/solutions/big-compute/`, 2015. [Online; accessed 13-December-2015].

[116] Google Developers. Architecture: High performance computing. `https://cloud.google.com/solutions/architecture/highperformancecomputing`, 2015. [Online; accessed 13-December-2015].

[117] Top500.org. Amazon web services — top500 supercomputer sites. `http://www.top500.org/site/50321`, 2015. [Online; accessed 13-December-2015].

[118] Hans Werner Meuer. The TOP500 project: Looking back over 15 years of supercomputing experience. *Informatik-Spektrum*, 31(3):203–222, 2008.

[119] Jack J. Dongarra, Piotr Luszczek, and Antoine Petite. The LINPACK benchmark: Past, present and future. *Concurrency Computation Practice and Experience*, 15(9):803–820, 2003.

[120] Inc. Amazon Web Services. Aws cloudformation - amazon web services. `https://aws.amazon.com/cloudformation/`, 2015. [Online; accessed 13-December-2015].

[121] Google Developers. Cloud deployment manager - google cloud platform. `https://cloud.google.com/deployment-manager/`, 2015. [Online; accessed 13-December-2015].

[122] Inc. Amazon Web Services. Amazon emr — amazon web services (aws) — big data processing. `https://aws.amazon.com/elasticmapreduce/`, 2015. [Online; accessed 13-December-2015].

[123] Google Developers. Cloud dataproc - google cloud platform. `https://cloud.google.com/dataproc/`, 2015. [Online; accessed 13-December-2015].

[124] E G Jr. Coffman, M.R. Garey, and David S. Johnson. Approximation Algorithms for Bin Packing: A Survey. *Approximation Algorithms*, pages 1–53, 1996.

[125] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[126] Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, 24(3):45–77, December 2007.

[127] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[128] Jacopo Urbani. *On Web-scale Reasoning*. PhD thesis, VRIJE UNIVERSITEIT, February 2013.

[129] Zoi Kaoudi. *Distributed RDF Query Processing and Reasoning in Peer-to-Peer Networks*. PhD thesis, National and Kapodistrian University of Athens, July 2011.

[130] Germán Moltó, Miguel Caballer, Eloy Romero, and Carlos de Alfonso. Elastic Memory Management of Virtualized Infrastructures for Applications with Dynamic Memory Requirements. *Procedia Computer Science*, 18:159–168, 2013.

[131] Steven Pigeon. *Contributions to data compression*. PhD thesis, Université de Montréal, December 2001.

[132] Java Archive (JAR). `http://docs.oracle.com/javase/6/docs/technotes/guides/jar/index.html`, 2015. [Online; accessed 13-December-2015].

[133] T Kaefer. NxParser - a collection of java parsers for different rdf serialisations. `https://github.com/nxparser/nxparser`, 2015. [Online; accessed 13-December-2015].

[134] Software Esoteric. Kryo - java serialization and cloning: fast, efficient, automatic. `https://github.com/EsotericSoftware/kryo`, 2015. [Online; accessed 13-December-2015].

[135] Software Esoteric. Benchmark comparing serialization libraries on the jvm. `https://github.com/eishay/jvm-serializers/wiki`, 2015. [Online; accessed 13-December-2015].

[136] Boanerges Aleman-Meza, Farshad Hakimpour, I. Budak Arpinar, and Amit P. Sheth. SwetoDblp ontology of Computer Science publications. *Web Semantics*, 5(3):151–155, 2007.

[137] Google Developers. Bucket locations - google cloud platform. `https://cloud.google.com/storage/docs/bucket-locations`, 2015. [Online; accessed 13-December-2015].

[138] Oracle Database Semantic Technologies Performance Summary for Sun M8000 Server . `http://download.oracle.com/otndocs/products/semantic_tech/pdf/semtech_performance_lubm8000_110819.pdf`, 2011. [Online; accessed 13-December-2015].

[139] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and F. van Harmelen. Scalable distributed reasoning using MapReduce. *The Semantic Web-ISWC 2009*, pages 634–649, 2009.

Appendices

Appendix A

ECARF Tasks And Job Definition

This Appendix contains a description of all the ECARF tasks and the end-to-end job definition JSON for these tasks.

A.1 ECARF Tasks

ECARF tasks are summarised in Table A.1, the table gives the task name and indicates if they are executed by the coordinator or processor. Additionally, the table indicates which of the tasks uses multiple threads internally, finally, the number of VMs used to execute the task is also provided. A brief description is provided for each of these tasks as follows:

ExtractSchemaTermTask: Load the schema file and extract the relevant terms.

CreateFileItemsTask: List the dataset files in the Cloud Storage folder and create partitions based on the number of processors. These partitions are added to the job context.

DummyProcessorTask: An empty task used to measure the time required to start the processors.

Table A.1: ECARF Tasks Summary

Task	**Executed By**	**Multi-threaded**	**Num. of VMs**
ExtractSchemaTermTask	Coordinator	N	1
CreateFileItemsTask	Coordinator	N	1
DummyProcessorTask	Processor	N	n
ExtractCountTerms2PartTask	Processor	Y	n
CombineTermStatsTask	Coordinator	N	1
AssembleDictionaryTask	Processor	Y	1
ProcessLoadTask	Processor	Y	n
LoadBigDataFilesTask	Coordinator	N	1
LoadEncodedTermStatsTask	Coordinator	N	1
DoReasonTask9	Processor	Y	n
DoUploadOutputLogTask	Coordinator Processor	N	n

ExtractCountTerms2PartTask: Each processor counts the instance triples for the relevant schema terms and extracts the URI parts for the dictionary. These are serialised into files and uploaded to Cloud Storage alongside a file that contains the count of instance triples.

CombineTermStatsTask: The coordinator combines the count files generated by the various processors into a TermStats file and uploads it Cloud Storage.

AssembleDictionaryTask: The processor downloads all the serialised URI parts from Cloud Storage and assembles the dictionary. The dictionary is then serialised to disk and uploaded to Cloud Storage. This task also encodes the schema and the TermStats files and uploads them to Cloud Storage.

ProcessLoadTask: Based on the file partitions created by the CreateFileItemsTask, each processor is assigned a number of files and download the dictionary from Cloud Storage. Each file is then encoded and serialised as CSV and uploaded to Cloud Storage.

LoadBigDataFilesTask: The coordinator loads all the CSV files created by the processors into a BigQuery table.

LoadEncodedTermStatsTask: The coordinator loads the encoded TermStats file from Cloud Storage and sets the encoded terms and their count on the job context.

DoReasonTask9: Each processor does forward reasoning using the relevant schema terms assigned to it.

DoUploadOutputLogTask: A generic task used to upload the output logs of a coordinator or a processor to Cloud Storage.

A.2 ECARF End-to-End Job Definition

The ECARF end-to-end job definition JSON is provided below. The job data is changed according to each of the datasets being processed.

```
{
  "id": "ecarf-forward-reasoning-lubm",
  "description": "Forward reasoning on the LUBM datasets
    ↪ using dictionary compression and 16 processors",
  "data": {
    "sourceBucket": "lubm3",
    "bucket": "lubm-fullrun-16n",
    "schema": "univ-bench.nt",
    "table": "lubm_tables.lubm16n",
    "termStatsFile": "term_stats.json",
    "encodedSchemaFile": "univ-bench_encoded.csv",
    "encodedTermStatsFile": "term_stats_encoded.json",
    "dictionaryFile": "lubm_dictionary.kryo.gz",
    "countOnly": "false",
    "encode": "true",
    "jobId": "run1",
    "splitLocation": "-1",
```

```
    "numberOfProcessors": 16
  },
  "vmConfig": {
    "zoneId": "us-central1-a",
    "imageId": "ecarf-1000/global/images/ecarf-debian-8-
      ↪ jessie-v20160219-8",
    "vmType": "n1-highmem-4",
    "networkId": "default",
    "diskType": "pd-standard",
    "diskSize": 100,
    "startupScript": "su - omerio -c 'cd /home/omerio/ecarf/
      ↪ ecarf-evm && export VM_XMS=\"-Xms512m\" VM_XMX=\"-
      ↪ Xmx23g\" && mvn -q exec:exec 2>&1 & exit 0' exit 0"
      ↪ ,
    "cost": 0.252,
    "minUsage": 600
  },
  "tasks": [
    {
      "id": "schema-term-extract-task",
      "description": "Load the schema and extract the
        ↪ relevant the relevant terms",
      "className": "io.ecarf.core.cloud.task.coordinator.
        ↪ ExtractSchemaTermTask",
      "input": {
        "sourceBucket": "#sourceBucket",
        "bucket": "#bucket",
        "schemaFile": "#schema"
      },
      "output": [
        "schemaTermsFile"
      ],
      "target": "COORDINATOR",
      "errorAction": "EXIT"
    },
    {
      "id": "partition-load-task",
      "description": "Find all the files in the cloud bucket
        ↪ and add them to a list of items",
      "className": "io.ecarf.core.cloud.task.coordinator.
        ↪ CreateFileItemsTask",
      "input": {
        "bucket": "#sourceBucket"
```

```
    },
    "output": [
      "fileItems"
    ],
    "target": "COORDINATOR",
    "errorAction": "EXIT"
  },
  {
    "id": "start-processors-task",
    "description": "A dummy task to start all the
      ↪ processors before hand",
    "className": "io.ecarf.core.cloud.task.processor.
      ↪ DummyProcessorTask",
    "target": "PROCESSOR",
    "errorAction": "EXIT",
    "partitioning": {
      "type": "COUNT",
      "countRef": "#numberOfProcessors"
    }
  },
  {
    "id": "extract-count-terms-task",
    "description": "Extract all the dictionary parts and
      ↪ count the relevant schema terms",
    "className": "io.ecarf.core.cloud.task.processor.
      ↪ analyze.ExtractCountTerms2PartTask",
    "input": {
      "sourceBucket": "#sourceBucket",
      "bucket": "#bucket",
      "schemaTermsFile": "#schemaTermsFile",
      "splitLocation": "#splitLocation",
      "files": "#filePartitions"
    },
    "target": "PROCESSOR",
    "errorAction": "EXIT",
    "partitioning": {
      "type": "FUNCTION",
      "functionName": "BinPackingPartition",
      "input": {
        "items": "#fileItems",
        "numberOfBins": "#numberOfProcessors"
      },
      "output": "filePartitions"
```

```
    }
  },
  {
    "id": "term-stats-task",
    "description": "Read and join up the term stats
      ↪ generated by the various processors",
    "className": "io.ecarf.core.cloud.task.coordinator.
      ↪ CombineTermStatsTask",
    "input": {
      "processors": "#processors",
      "bucket": "#bucket",
      "termStatsFile": "#termStatsFile"
    },
    "output": [
      "termItems"
    ],
    "target": "COORDINATOR",
    "errorAction": "EXIT"
  },
  {
    "id": "assemble-dictionary-task",
    "description": "Assemble the terms dictionary",
    "className": "io.ecarf.core.cloud.task.processor.
      ↪ dictionary.AssembleDictionaryTask",
    "input": {
      "bucket": "#bucket",
      "targetBucket": "#bucket",
      "schemaBucket": "#sourceBucket",
      "schemaFile": "#schema",
      "termStatsFile": "#termStatsFile",
      "encodedSchemaFile": "#encodedSchemaFile",
      "encodedTermStatsFile": "#encodedTermStatsFile",
      "dictionaryFile": "#dictionaryFile"
    },
    "target": "PROCESSOR",
    "errorAction": "EXIT",
    "partitioning": {
      "type": "COUNT",
      "count": 1
    }
  },
  {
    "id": "process-load-task",
```

```
    "description": "Each processor encodes and converts the
      ↪  files assigned to it to CSV",
    "className": "io.ecarf.core.cloud.task.processor.
      ↪ ProcessLoadTask",
    "input": {
      "sourceBucket": "#sourceBucket",
      "bucket": "#bucket",
      "countOnly": "#countOnly",
      "encode": "#encode",
      "dictionaryFile": "#dictionaryFile",
      "schemaTermsFile": "#schemaTermsFile",
      "files": "#filePartitions"
    },
    "target": "PROCESSOR",
    "errorAction": "EXIT",
    "partitioning": {
      "type": "FUNCTION",
      "functionName": "BinPackingPartition",
      "input": {
        "items": "#fileItems",
        "numberOfBins": "#numberOfProcessors"
      },
      "output": "filePartitions"
    }
  },
  {
    "id": "big-data-load-task",
    "description": "Load all the files processed by the
      ↪ processors into BigQuery",
    "className": "io.ecarf.core.cloud.task.coordinator.
      ↪ LoadBigDataFilesTask",
    "input": {
      "table": "#table",
      "bucket": "#bucket",
      "encode": "#encode"
    },
    "target": "COORDINATOR",
    "errorAction": "EXIT"
  },
  {
    "id": "encoded-term-stats-load-task",
    "description": "Load the encoded terms stats into the
      ↪ job context",
```

```
    "className": "io.ecarf.core.cloud.task.coordinator.
      ↪ LoadEncodedTermStatsTask",
    "input": {
      "bucket": "#bucket",
      "encodedTermStatsFile": "#encodedTermStatsFile"
    },
    "output": [
      "termItems"
    ],
    "target": "COORDINATOR",
    "errorAction": "EXIT"
  },
  {
    "id": "process-reason-task",
    "description": "Distribute the reason task between the
      ↪ various processors, each processor then reasons
      ↪ over the terms assigned to it",
    "className": "
      ↪ io.ecarf.core.cloud.task.processor.reason.
      ↪ phase2.DoReasonTask9",
    "input": {
      "bucket": "#bucket",
      "table": "#table",
      "schemaFile": "#encodedSchemaFile",
      "terms": "#termPartitions"
    },
    "target": "PROCESSOR",
    "errorAction": "CONTINUE",
    "partitioning": {
      "type": "FUNCTION",
      "functionName": "BinPackingPartition1",
      "input": {
        "items": "#termItems",
        "numberOfBins": "#numberOfProcessors"
      },
      "output": "termPartitions"
    }
  },
  {
    "id": "processor-do-upload-logs-task",
    "description": "Distribute the uploading of logs task
      ↪ between the various processors, each processor
      ↪ then uploads its own logs to cloud storage",
```

```
      "className": "io.ecarf.core.cloud.task.common.
        ↪ DoUploadOutputLogTask",
      "input": {
        "bucket": "#bucket",
        "jobId": "#jobId"
      },
      "target": "PROCESSOR",
      "errorAction": "CONTINUE",
      "partitioning": {
        "type": "ITEMS",
        "input": {
          "items": "#processors"
        }
      }
    },
    {
      "id": "coordinator-do-upload-logs-task",
      "description": "The coordinator to upload its own logs
        ↪ to cloud storage",
      "className": "io.ecarf.core.cloud.task.common.
        ↪ DoUploadOutputLogTask",
      "input": {
        "bucket": "#bucket",
        "jobId": "#jobId"
      },
      "target": "COORDINATOR",
      "errorAction": "CONTINUE"
    }
  ]
}
```

Appendix B

CloudEx and ECARF Source Code

The CloudEx and ECARF Java source code can be found in the accompanying CD or at `http://cloudex.io` and `http://ecarf.io`.