Distributed Rule-Based Ontology Reasoning

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

by

Raghava Mutharaju
B.Tech., Jawaharlal Nehru Technological University, 2004

2016
Wright State University
I HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER MY SUPERVISION BY Raghava Mutharaju ENTITLED Distributed Rule-Based Ontology Reasoning BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy.

Pascal Hitzler, Ph.D.
Dissertation Director

Michael L. Raymer, Ph.D.
Director, Computer Science and Engineering Ph.D. Program

Robert E.W. Fyffe, Ph.D.
Vice President for Research and Dean of the Graduate School

Committee on Final Examination

Pascal Hitzler, Ph.D.

Prabhaker Mateti, Ph.D.

Derek Doran, Ph.D.

Freddy Lécué, Ph.D.

Frederick Maier, Ph.D.
The vision of the Semantic Web is to provide structure and meaning to the data on the Web. Knowledge representation and reasoning play a crucial role in accomplishing this vision. OWL (Web Ontology Language), a W3C standard, is used for representing knowledge. Reasoning over the ontologies is used to derive logical consequences. A fixed set of rules are run on an ontology iteratively until no new logical consequences can be derived. All existing reasoners run on a single machine, possibly using multiple cores. Ontologies (sometimes loosely referred to as knowledge bases) that are automatically constructed can be very large. Single machine reasoners will not be able to handle these large ontologies. They are constrained by the memory and computing resources available on a single machine.

In this dissertation, we use distributed computing to find scalable approaches to ontology reasoning. In particular, we explore four approaches that use a cluster of machines for ontology reasoning – 1) A MapReduce approach named MR-EL where reasoning happens in the form of a series of map and reduce jobs. Termination is achieved by eliminating the duplicate consequences. The MapReduce approach is simple, fault tolerant and less error-prone due to the usage of a framework that handles aspects such as communication, synchronization etc. But it is very slow and does not scale well with large ontologies. 2) Our second approach named DQuEL is a distributed version of a sequential reasoning algorithm used in the CEL reasoner. Each node in the cluster applies all of the rules and
generates partial results. The reasoning process terminates when each node in the cluster has no more work to do. DQuEL works well on small and medium sized ontologies but does not perform well on large ontologies. 3) The third approach, named DistEL, is a distributed fixpoint iteration approach where each node in the cluster applies only one rule to a subset of the ontology. This happens iteratively until all of the nodes cannot generate any more new logical consequences. This is the most scalable of all of the approaches. 4) Our fourth approach, named SparkEL, is based on the Apache Spark framework where each reasoning rule is translated into a form that is suitable for Apache Spark. Several algorithmic and framework related optimizations were considered. SparkEL works very well on small and medium sized ontologies, but it does not scale to large ontologies. All four distributed reasoning systems work on a subset of OWL 2 EL which is a tractable profile of OWL with a polynomial reasoning time. Along with the description of the algorithms, optimizations and evaluation results of the four distributed reasoners, we also provide recommendations for the best choice of reasoners for different scenarios.
# Contents

## 1 Introduction

1.1 Semantic Web .................................................. 1
   1.1.1 RDF and OWL .............................................. 1
   1.1.2 Description Logics ...................................... 2
   1.1.3 Reasoning .................................................. 3
   1.1.4 OWL 2 Profiles .......................................... 4

1.2 Large Ontologies .............................................. 6

1.3 Research Statement ........................................... 7

1.4 Challenges ..................................................... 8

1.5 Contributions .................................................. 9

1.6 Organization ................................................... 10

## 2 Background

2.1 Description Logics ............................................. 12
   2.1.1 Ontology Normalization ................................. 14
   2.1.2 Completion Rules ....................................... 15

2.2 Distributed Computing Models ............................... 17
   2.2.1 MapReduce ............................................... 17
   2.2.2 Peer-to-Peer Computing ................................. 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.3</td>
<td>Resilient Distributed Datasets</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Message Passing</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Key-Value Stores</td>
<td>24</td>
</tr>
<tr>
<td>2.5</td>
<td>Datasets</td>
<td>26</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Ontology Complexity</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>MapReduce Approach</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>Parallelization using MapReduce</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Theoretical Analysis</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Evaluation</td>
<td>40</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Strengths</td>
<td>41</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Runtime Inefficiency</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>Distributed Queue Approach</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>Queue Algorithm</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Distributed Queue Approach</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Evaluation</td>
<td>47</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Strengths</td>
<td>49</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Runtime Inefficiency</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Distributed Fixpoint Iteration Approach</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Distributed Fixpoint Iteration Algorithms</td>
<td>54</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Partitioning the Ontology</td>
<td>55</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Rule Processes</td>
<td>57</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Service Processes UN and RN</td>
<td>58</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Retrieval of Axioms from the Key Value Store</td>
<td>59</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Termination</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>Optimizations</td>
<td>61</td>
</tr>
</tbody>
</table>
5.3 Work Stealing .................................................. 62
5.4 Evaluation ..................................................... 64
  5.4.1 Test Data .................................................. 64
  5.4.2 Results .................................................... 65
  5.4.3 Discussion ............................................... 67
  5.4.4 Runtime Inefficiency ................................. 68

6 Spark based Approach .................................. 78
  6.1 Axiom Representation ................................. 79
  6.2 Algorithms ............................................... 80
    6.2.1 Pseudocode for Completion Rules .......... 81
  6.3 Optimizations .......................................... 87
  6.4 Implementation Details .............................. 88
    6.4.1 Spark Framework related Optimizations ... 89
  6.5 Evaluation .............................................. 92
    6.5.1 Evaluation Goals ................................. 92
    6.5.2 Datasets .............................................. 93
    6.5.3 Results .............................................. 93
    6.5.4 Discussion .......................................... 95

7 Comparison of Approaches ............................. 99
  7.1 Flynn’s Taxonomy Revisited ......................... 99
  7.2 Characteristics of the four distributed reasoners .. 101
  7.3 Recommendations ...................................... 106
  7.4 The Way Forward ...................................... 109

8 Related Work ............................................. 111
  8.1 Scalable Reasoning ..................................... 111
List of Figures

1.1 OWL 2 Profiles. Image obtained from https://www.w3.org/TR/2009/WD-owl2-overview-20090327/ ....................... 5

2.1 An example $\mathcal{EL}^+$ ontology .......................................................... 13

2.2 Map and Reduce operations ................................................................. 18

2.3 Driver and worker nodes in Apache Spark. Redrawn image from http://spark.apache.org/docs/latest/cluster-overview.html 21

3.1 Scalability of MapReduce approach on 8 nodes with different copies of GALEN ontology ................................................. 41

4.1 Scalability of DQuEL on 13 nodes with different ontologies ............. 51

5.1 Node assignment to rules and dependency among the completion rules. A rectangle represents a node in the cluster and inner ovals represent sub-clusters. Outer ovals enclosing $SC_1/SC_2$, and $SC_7/SC_8$ show their tighter input-output relationships. The set (U[X], R[r], or Q[r]) affected by the rule is shown within the enclosing oval. For simplicity, only one node is shown to hold results. ..................................................... 56

5.2 Scalability of DistEL on a cluster of nodes with different ontologies .... 66

5.3 Scalability of DistEL on a cluster of nodes with the Traffic ontology .... 66
6.1 Reasoner runtime in seconds for DistEL and SparkEL on various ontologies (smaller is better). These are run on a 4 node cluster ........................... 96
6.2 Reasoner runtime in seconds for DistEL and SparkEL on SNOMED CT with different cluster size .............................................................. 97
6.3 Memory management in Apache Spark. Image obtained from https://0x0fff.com/spark-memory-management/ ....................... 98

7.1 Single Rule, Multiple Data architecture .............................................. 100
7.2 Multiple Rule, Multiple Data architecture .......................................... 101

10.1 RDFS rule dependency graph ............................................................ 124
10.2 Rules can be divided into S and R groups based on their output .......... 126
10.3 Flowchart that indicates the termination procedure of $S_g$ and $R_g$ .... 127
List of Tables

2.1 The classification completion rules for $\mathcal{EL}^{++}$. Rules $\textbf{R7} - \textbf{R9}$ in [6] deal with concrete domains and so are not included. ........................................ 16

2.2 The classification completion rules for $\mathcal{EL}^+$. .................................................. 16

2.3 Various ontologies and their availability ................................................................. 27

3.1 Revised $\mathcal{EL}^{++}$ algorithm. In R10, $r$ is allowed to be of the form $r_1 \circ r_2$. The keys are used in the MapReduce algorithm presented in Section 3.1. ........ 32

3.2 Classification time (in seconds) of MapReduce (MR) approach. OOM indicates Out Of Memory. .............................................................. 41

4.1 Key-Value pairs for axioms .............................................................. 46

4.2 Classification time (in seconds) of Pellet, jCEL and ELK reasoners. OOM stands for Out Of Memory ......................................................... 50

4.3 Classification time (in seconds) of DQuEL ................................................. 50

5.1 Completion Rules and Key Value Pairs ..................................................... 53

5.2 Number of axioms, before and after classification, in ontologies ............ 76

5.3 Classification times in seconds. OOM$^a$: reasoner runs out of memory. OOM$^b$: reasoner runs out of memory during incremental classification.

OOM$^c$: ontology too big for OWL API to load in memory. .............. 76

5.4 Classification time (in seconds) of DistEL .............................................. 76
5.5 Memory taken by Redis on each node for traffic data

5.6 Speedup achieved by ELK, with all the threads on one 8-core machine, on SNOMED CT

5.7 Speedup achieved by DistEL on SNOMED CT

5.8 Speed (in seconds) for simple read, write operations of 1,000,000 items using RAM and Redis

6.1 The classification completion rules for $\mathcal{EL}^+$.  
6.2 Axiom format for different axiom types

6.3 Number of axioms in ontologies, before and after reasoning.

6.4 Reasoning times in seconds of various popular single machine reasoners. OOM is out of memory error. ERR stands for parse error.

6.5 Reasoning times in seconds of distributed reasoners, DistEL and SparkEL. OOM$^a$ represents out of memory on the worker (slave) nodes and OOM$^b$ represents out of memory on the driver (master) node.

7.1 Characteristics of the four distributed reasoners discussed in this dissertation

7.2 Recommendations for a reasoner on single machine based on the ontology size and the machine type available for reasoning

7.3 Recommendations for a reasoner on a shared-nothing cluster based on the ontology size and the machine type available for reasoning

8.1 Some popular NoSQL stores along with their data model and compliance with CAP theorem

10.1 RDFS closure rules

A.1 Completion Rules for classification
List of Algorithms

1. map for word count ......................................................... 19
2. reduce for word count ....................................................... 19
3. Word count in Apache Spark ............................................... 22
4. replicateOntology(ontologyURI, n) ..................................... 29
5. MapReduce algorithm for R2-1 ........................................... 36
6. MapReduce algorithm for R2-2 ........................................... 37
7. MapReduce algorithm for R6-3-2. The input of the map function is an axiom, taken from \(O, S, P, R, H, J\), or \(N\). Key-value pairs are generated in the map phase, and these serve as input in the reduce phase. For a given key, every possible combination of values is examined to determine whether R6-3-2 is applicable. Ultimately, a list of axioms is produced as output. . . . 38
8. MapReduce algorithm for R6-3-3 ........................................... 39
9. process\((A, X)\) .......................................................... 45
10. process-new-edge\((A, r, B)\) ............................................... 46
11. Termination Controller, TC ................................................. 48
12. OneIteration() ............................................................. 49
13. Job Controller ............................................................... 49
14. General fixpoint iteration strategy for applying rules ............... 54
Algorithm for node assignment i.e., from among the available number of nodes, this algorithm calculated the number of nodes to be assigned to each rule/ontology fragment. .......................................................... 70

R1: \( A \subseteq B \Rightarrow U[B] \subseteq U[A] \) ............................................. 71

R2: \( A_1 \cap \cdots \cap A_n \subseteq B \Rightarrow U[B] \models U[A_1] \cap \cdots \cap U[A_n] \) 71

R3: \( A \subseteq \exists r. B \Rightarrow R[r] \models \{(X, B) \mid X \in U[A]\} \) ............. 71

R4: \( \exists r. A \subseteq B \Rightarrow Q[r] \models \{(Y, B) \mid Y \in U[A]\} \) ............. 71

R5: \((X, Y) \in R[r] \land (Y, B) \in Q[r] \Rightarrow U[B] \models \{X\} \) ................. 72

R6: \( X \subseteq \exists r. Y \Rightarrow U[\bot] \models \{X \mid Y \in U[\bot]\} \) ................. 72

R7: \( r \subseteq s \Rightarrow R[s] \models R[r] \) ......................................................... 72

R8: \( r \circ s \subseteq t \Rightarrow R[t] \models \{(X, Z) \mid (X, Y) \in R[r], (Y, Z) \in R[s]\} \) .... 73

Process UN maintains \( U[X] \), for all \( X \). .................................................. 73

Node RN maintains \( R[r] \) sets ................................................................. 73

Wrapper for Ri ............................................................... 74

Wrapper for UN and RN ............................................................... 74

Algorithm to process axioms with work stealing ........................................ 74

Algorithm to divide the set of axioms into chunks .................................... 74

Algorithm to retrieve a work chunk ...................................................... 75

Algorithm to process one chunk of work ............................................. 75

Algorithm to steal the work from the busiest node ................................ 76

R1: \( A \subseteq B, A \in S(X) \Rightarrow S(X) \models \{B\} \) ................................. 82

R2: \( A_1 \cap A_2 \subseteq B, A_1, A_2 \in S(X) \Rightarrow S(X) \models \{B\} \) ............. 83

R3: \( A \subseteq \exists r. B, A \in S(X) \Rightarrow R(r) \models \{(X, B)\} \) .................... 83

R4: \((X, Y) \in R(r), A \in S(Y), \exists r. A \subseteq B \Rightarrow S(X) \models \{B\} \) ........ 84

R5: \((X, Y) \in R(r), r \subseteq s \Rightarrow R(s) \models \{(X, Y)\} \) .......................... 85

R6: \((X, Y) \in R(r_1), (Y, Z) \in R(r_2), r_1 \circ r_2 \subseteq r_3 \Rightarrow R(r_3) \models \{(X, Z)\} \) 86

Wrapper algorithm that applies all the rules on the axioms until completion 87
customizedDistinct(List[(Int, Int)] partitions) 

customizedSubtract(List[(Int, Int)] partitions1, List[(Int, Int)] partitions2)
Acknowledgments

Many supported me either directly or indirectly and I got to learn from several people. I thank my Ph.D. supervisor Prof. Pascal Hitzler for his support and patience. It has been a long journey and without his support it would not have been possible to complete the Ph.D. journey. It is always a pleasure to work with people who are modest, kind and fair and Pascal certainly falls into that category.

Apart for my advisor, another faculty member from the Computer Science department with whom I spent considerable amount of time is Prabhaker Mateti. We spent long hours discussing not only technical things but several non-technical things as well. Conversations with him have always been interesting and jovial. His course in Distributed Computing Principles attracted me towards Distributed Computing and also to make it a part of my dissertation study. I thank him for all our conversations.

I would like to thank rest of the committee members Dr. Derek Doran, Dr. Frederick Maier and Dr. Freddy Lécué for their valuable comments and suggestions to improve my dissertation work. I also thank Prof. Amit Sheth for our work collaborations during my first year of Ph.D. I learnt several important things either directly or indirectly from him. I would like to thank Prof. T.K. Prasad for our pleasant conversations.

My family has been a constant source of support for me. I thank my parents, Gopal Rao and Padmavathi and my sister Jyothsna for their patience and support. I thank my wife, Lalitha Sravanthi for her love, patience, support and for putting up with me when I couldn’t spend sufficient time with her during my Ph.D.

There are lots of ups and downs in a Ph.D. journey and it would not have been possible to complete it without the support of my friends and colleagues. I thank Pramod Anantharam, Archana, Pavan Kapanipathi, Revathy, Ashutosh Jadhav, Rashmi, Hemant Purohit, Harshal Patni, Pritish, Rahul Gazula, Amogh, Ramakanth Kavuluru, Wenbo Wang, Vinh Nguyen, Sanjaya, Sujan, Sarasi, Lu Chen, Shreyansh, Lakshika, Kalpa, Sreeram, Hima Bindu, Delroy Cameron, Karthik Gomadam, Meena Nagarajan, Pablo Mendes, Cory Hen-
son, Ajith Ranabahu, Christopher Thomas (Topher), Prateek Jain, Matthew Perry, Cartic Ramakrishnan, and Satya Sahoo for their help, support and guidance during my Ph.D.

The Data Semantic Lab (DaSe) lab has been a great place to work and I would like to thank all my friends and colleagues at the lab. It was a great work environment. Special thanks to Adila Krisnadhi, Kunal Sengupta, Amit Joshi, Cong Wang (Joshua), David Carral, Reihaneh Amini, Kamruzzaman Sarker, Nazifa Karima, Lu Zhou, and Michelle Cheatham. I also thank the visitors of DaSe lab, Sambhawa Priya, Andreas Thalhammer, Victor, Seif, Ilias and Fabian for our pleasant conversations.

I had the opportunity to do summer internships at many great places. Internships were a nice distraction and helped me to visit new places and meet new people. I thank my first internship supervisor Dr. Evren Sirin from Clark & Parsia (now Complexible Inc.) for providing valuable suggestions. I thank my internship team, Michael Shepherd, Kirk Ocke and Dale Gaucas from Xerox Research Center, Webster (XRCW) for their support during my internship. It was a pleasant experience working at XRCW and a much needed diversion from my Ph.D. work. I thank my internship supervisors Dr. Sherif Sakr and Dr. Alessandra Sala from Bell Labs. My internship at IBM Research, Ireland was very helpful to my Ph.D. work and lead to more collaborations. For this, I would like to thank my internship supervisor Dr. Freddy Lécué. I would also like to thank Geeth R De Mel, my internship supervisor at IBM T.J. Watson Research Center.

The administrative staff in the CS department have always been helpful. I would like to thank Tonya Davis, Paula Price, and Diane Slusser for helping me with all the paper work.

I thank all the funding sources during my doctoral study. This Ph.D. is partially funded by the National Science Foundation (NSF) project 1017225 “III: Small: TROn – Tractable Reasoning with Ontologies”. I thank Microsoft Azure sponsorship and Amazon AWS Education grant for providing the funding computational resources. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding sources.
Dedicated to

my parents Gopal Rao and Padmavathi

and to my wife Lalitha Sravanthi for their patience and support
1 Introduction

1.1 Semantic Web

The World Wide Web is an interlinked collection of documents that are generally devoid of any structure and primarily meant for human consumption. On the other hand, the Semantic Web [11] is an extension to the World Wide Web where structure and meaning are provided to the data which in turn makes the information in the interlinked documents machine understandable. This also enables the interlinking, sharing and reuse of data on the Web. The goal of the Semantic Web is to go from the current Web of documents towards the Web of data. In order to provide the structure, meaning and context to the data, knowledge of the domain is essential. The two W3C standards that play a key role in the realization of the Semantic Web vision are RDF, which stands for Resource Description Framework [65], and OWL, which stands for Web Ontology Language [41, 42].

1.1.1 RDF and OWL

RDF is a framework to describe resources on the Web such as documents, people, physical objects, abstract concepts, strings, numbers etc. It can be used for interlinking various datasets. Resources are described using triples and are identified by IRIs (International Resource Identifiers). RDF triples consist of three components: subject, predicate and the object. A triple implies that the relationship indicated by the predicate holds between the
two resources represented by the subject and object. An example is given below.

\[ \text{<John> <isUserOf> <Facebook>} \]
\[ \text{<John> <hasAccountOn> <LinkedIn>} \]
\[ \text{<John> <uses> <Twitter>} \]

This example indicates that John has profiles on three different social networking platforms. Since the resource John would have the same IRI across the datasets of the three social networks, it would be easy to interlink the data from all the three [66]. A set of triples is nothing but a directed labeled graph with the subject and object as the two nodes and predicate as the edge label. This graph can be queried using a query language called SPARQL [38], a W3C standard. It performs subgraph matching to compute the query results. Although RDF provides a simple and effective way to represent relationships between the resources, it cannot be used for complex relationships. OWL has more expressivity and can be used to represent complex relationships.

OWL is a Semantic Web language used to build ontologies which in turn represent knowledge about things, groups of things and relation between them. It is used to formally encode domain knowledge, i.e., knowledge about some part of the world which is often referred to as the domain of interest. In order to build an ontology, it is important to come up with the vocabulary of the domain, i.e., a set of terms and the relationships between them. These form the axioms in an ontology. The knowledge in an ontology can be categorized into terminological knowledge and assertions. The terminological knowledge or TBox defines the general notions or the conceptualization of the domain whereas the assertional knowledge or ABox defines the concrete notions or facts of the domain. In a database setting, TBox corresponds to the schema and ABox corresponds to the data [7].

1.1.2 Description Logics

Knowledge representation [21] is a field of Artificial Intelligence that encodes the knowledge of the domain in a form that is suitable to be processed by computer systems. There
are several knowledge representation languages such as first order logic, Semantic Networks [92], Frames [69], controlled natural language etc. The one that is of interest to us is Description Logics [55, 7] since it provides the underpinnings for OWL. Description logics are fragments of first order logic, with most of them being decidable. They have formal semantics, i.e., a precise specification of the constructs that make up various description logics. This makes them unambiguous and suitable for logical operations. Description logics provide three types of entities: concepts, roles and individual names. Concepts are sets of individuals, roles represent the binary relations between the individuals and individual names represent single individuals in the domain. In first order logic, these three entities correspond to unary predicates, binary predicates and constants. In OWL, concepts and roles are referred to as classes and properties.

1.1.3 Reasoning

The term knowledge base is often associated with models having different levels of semantics, starting with a loose vocabulary of terms and going up to a formal OWL model. In this dissertation, we consider RDF and OWL models as knowledge bases. Reasoning is one of the important operations that can be performed over a knowledge base (ontology). Since OWL is a declarative language, it does not specify how reasoning should be done. This task is performed by the reasoner. Reasoning is required to infer logical consequences and check the consistency of the knowledge base. These are two standard reasoning tasks. The reasoning task that is considered here is called classification, which is the computation of the complete subclass relation hierarchy for each concept in the ontology. Consider the following related facts.

1. John is a US citizen.

2. Someone who is a US citizen, has a US passport.

3. Someone who has a US passport does not require EU visa.
After reasoning on these facts, among other things, the following logical consequences are derived

(a) Each US citizen does not require an EU visa.

(b) John does not require an EU visa.

After deriving these logical consequences, it is easy to answer questions such as “does John require EU visa?” which cannot be answered based on just the given three facts. From the three given facts, it is trivial for a human to come up with the derived conclusions. A reasoner allows a machine to not only automatically derive the conclusions from the given knowledge but also do it at a larger scale. There are currently several well-known and efficient reasoners such as ELK [48], Konclude [93], Pellet [90], HermiT [31], FaCT++ [99], Snorocket [67] etc. Reasoners use either tableau algorithms which involve constructing models [10] or use inference rules to derive logical consequences [89]. In this dissertation, we consider the rule-based approach where we apply a set of inference rules on the given ontology until no new logical consequence can be derived. This is known as fixpoint iteration.

1.1.4 OWL 2 Profiles

With the focus on tractability and scalability, the most recent version of OWL, referred to as OWL 2, provides three profiles or fragments, namely, OWL 2 EL, OWL 2 QL and OWL 2 RL. Each profile offers different expressivity features and hence is useful in different application scenarios. They provide different aspects of expressivity in order to obtain different computational and implementation benefits. None of these profiles is a subset of each other, but there is some overlap among them as indicated in Figure 1.1.

OWL 2 EL is useful in domains where there are complex descriptions and very large number of concepts and relations. The description logic underlying this profile is $\mathcal{EL}^{++}$
This profile provides existential quantification and hence the name EL. Several large biomedical ontologies such as SNOMED-CT, NCI thesaurus, Galen, etc., are in this profile. It can also be used in modeling other complex domains such as system configurations, product inventories, and other scientific domains. Compared to the other two profiles, this profile has comparatively expressive class expressions. This profile also has a fairly expressive property expressions, including property chains. But it doesn’t support negation, disjunction, universal quantification or inverse properties. All the standard reasoning tasks take polynomial time with respect to the size of the ontology.

OWL 2 QL is useful in applications that have large instance data (ABox) and where the primary reasoning task is query answering. This profile is suitable for representing database schemas (entity relationships). Different databases can be integrated by designing a higher level ontology. The process of using an ontology to describe, query and integrate databases is called ontology-based data access (OBDA) [52]. Queries, referred to as conjunctive queries [49], written with respect to the ontology can be rewritten to a relational query language. This is the basis for the profile’s name. Sound and complete conjunctive query answering can be performed in LOGSPACE (more precisely AC$^0$) with respect to the size of the data (assertional knowledge). The description logic underlying this profile is DL-Lite [16]. OWL 2 QL does not allow existential quantification of roles to a class expression. But unlike OWL 2 EL, it supports inverse properties.

The OWL 2 RL profile is suitable for enriching RDF data with rules. It also allows for
scalable reasoning without giving up too much of expressivity. Reasoning in this profile can be implemented using a standard rule language. This is the reason behind naming this profile OWL 2 RL. The design of this profile is inspired from Description Logic Programs (DLP) [34] and pD* [97]. All the standard reasoning tasks can be performed in polynomial time with respect to the size of the ontology. Among other things, existential quantification is not supported in this profile.

1.2 Large Ontologies

Ontologies can be constructed either manually or automatically. Manually constructed ontologies are not too large. SNOMED CT\(^1\) is the largest known biomedical ontology with 430,844 axioms. Current laptops and machines with commodity hardware can efficiently reason over ontologies which are of the size of SNOMED CT. Although manually constructed ontologies are clean and useful in narrow domains, in order to scale, automated construction of ontologies is required.

Automated knowledge base construction and population (popularly referred to as KBC) is an active area of research where techniques from natural language processing, machine learning, information extraction etc., are used to build knowledge bases [78]. Several large knowledge bases have been constructed as part of academic projects such as DBpedia [59], YAGO [62], NELL [70], Wikidata [104] and also in the industry [36, 25]. Knowledge is extracted from text [70, 25] and streaming data [57]. These knowledge bases keep evolving and increase in size over time. NELL has already accumulated over 80 million facts (as of 2015) in its knowledge base and Google’s Knowledge Vault has around 1.6 billion facts (as of 2014). DBpedia has around 3 billion triples in 2014 and Wikidata currently has around 875 million triples. Around 7 million OWL axioms per day are generated from traffic data in Dublin city, Ireland. Existing reasoners cannot handle such large knowledge

\(^1\)http://www.ihtsdo.org/snomed-ct
bases. They are constrained by the resources available on a single machine.

Reasoning over large knowledge bases needs a scalable approach because it is not possible for a single machine to keep up with the growth rate of data. Distributed computing consists of a set of cooperating processes, possibly spread across a group of networked machines, which can interact with each other by passing messages in order to accomplish a common goal [2]. Memory is not shared among the machines. More machines can be added to the cluster in order to scale the memory and computational resources to the size of the data. Distributed systems offer a viable computing model for scalable reasoning.

1.3 Research Statement

Scalability of a system is the ability to perform gracefully as the amount of work it has to handle increases (load scalability) and also to accommodate enlargement in its structure without any major modifications to its architecture (structural scalability) [13].

The overall goal of this dissertation is to come up with a scalable approach that supports load and structural scalability for reasoning over OWL 2 EL ontologies. The hypothesis of our work is as follows:

Distributed computing principles and techniques can be used to scale rule-based reasoning algorithms of description logic OWL 2 EL in such a way that

(a) given sufficient number of nodes, reasoning on a large ontology always completes

(b) with the addition of nodes, there is a speedup in the runtime

We chose OWL 2 EL for our work due to the following reasons.

1. Several real-world biomedical ontologies are in this profile. For example, GO, NCI, FMA, SNOMED CT, and several others in NCBO, OBO Foundry repositories.

2. Reasoning in this profile is a polynomial time algorithm. No other fragment in OWL
has a reasoning time lower than this. On the other hand, there are OWL fragments whose reasoning time is exponential.

3. Reasoning on large ontologies in OWL 2 EL profile consumes quite a bit of memory [28]. This makes it suitable to evaluate distributed reasoning on large ontologies.

4. The number of rules required for reasoning are small (6 to 11), so it is not overly complicated to implement the reasoning rules.

5. There is already prior work in scalable reasoning over profiles such as RDF, OWL 2 RL and OWL 2 QL (see Section 8 for more details).

### 1.4 Challenges

There are several challenges in designing and building a distributed $\mathcal{EL}^+$ reasoner.

1. The $\mathcal{EL}^+$ reasoning algorithm is not embarrassingly parallel. In order to complete the reasoning task there will be some data exchanged among the nodes in the cluster. In this scenario, the following play a crucial role.

   - The ontology should be partitioned in such a way that the data exchange among the nodes is minimized
   - For an embarrassingly parallel problem, the computation performed on one node does not affect the computation on another node. Because of this, identifying a point at which to terminate is easy. But in the case of distributed $\mathcal{EL}^+$ reasoning, nodes in the cluster should co-ordinate to determine termination.

2. Reasoning is the process of inferring logical consequences. So the ontology size increases during the reasoning process. Depending on the ontology, it could increase by a factor of 15. This affects the load balance of the cluster during runtime. So an effective dynamic load balancing strategy is very important for a distributed reasoner.
3. There are several ways in which scalable algorithms can be designed. There are also several software tools and frameworks that can be used to implement the scalable algorithms. The choice of a tool or a framework could very well affect the design of an algorithm. But it is hard to predict which of these choices would lead to the best possible solution unless it is implemented and evaluated. Depending on the choice taken, implementing a distributed system in comparison to its equivalent sequential version is time consuming and error-prone.

1.5 Contributions

1. MapReduce algorithms for all the $\mathcal{EL}^+$ inference rules are proposed, implemented and evaluated (Chapter 3). Along with this, MapReduce algorithms for $\mathcal{EL}^{++}$ reasoning are also proposed. Axioms are represented in a format suitable for MapReduce. Evaluation results are discussed where it is shown that this approach is slow and does not scale well with large ontologies.

2. A distributed version of the sequential classification algorithms of CEL\(^2\) reasoner are proposed, implemented and evaluated (Chapter 4). Detecting termination in a distributed setting is not straightforward. A termination algorithm using a central process is proposed along with the distributed reasoning algorithms. Evaluation results are discussed. It is shown that this approach works very well for small to medium size ontologies, but does not scale well to large ontologies.

3. Distributed reasoning algorithms for the so-called naive fixpoint iteration approach are proposed, implemented and evaluated (Chapter 5). Along with the reasoning algorithms, several optimizations such as work stealing, selective replication of axioms and techniques to avoid processing of unnecessary axioms have been proposed.

\(^2\)http://lat.inf.tu-dresden.de/systems/cel
Effective ontology partitioning strategy to reduce inter-node communication is proposed. A peer-to-peer termination detection algorithm has been proposed and discussed in detail. The distributed reasoning system, named DistEL is compared with other state-of-the-art reasoners and it is shown that DistEL scales well with the size of the ontologies.

4. Apache Spark\(^3\) based reasoning algorithms are proposed, implemented and evaluated (Chapter 6). Pseudocode for all the rule implementation along with an example for each is discussed. Several algorithmic optimizations such as join ordering, pruning join operands, key ordering, etc., are proposed. Apache Spark framework related optimizations such as caching, harsh partitioning, and broadcasting the data are also proposed. This distributed reasoning approach is evaluated on large ontologies and the results are compared with the state-of-the-art reasoners.

5. All the four distributed reasoning approaches are compared in Chapter 7. The similarities and differences among the four approaches are discussed in detail. Recommendations on reasoners to use for particular scenarios are given.

6. Alternate actor-based distributed reasoning approach named ELDer is proposed and discussed in detail in Chapter 10. A general framework for rule-based distributed reasoning, that can handle any given ruleset is also discussed.

### 1.6 Organization

This dissertation is organized as follows: Chapter 2 introduces the background material that is required for the dissertation. It contains brief descriptions of the description logics $\mathcal{EL}^+$ and $\mathcal{EL}^{++}$ as well as some distributed computing concepts and frameworks. Chapter 3 describes the MapReduce based $\mathcal{EL}^+$ reasoner including the pseudocode for all of the algo-

\(^3\)http://spark.apache.org/
Chapter 4 describes DQuEL, a distributed queue based $\mathcal{EL}^+$ reasoner including the pseudocode for all of the algorithms and the experiment results. Chapter 5 describes another approach to distributed $\mathcal{EL}^+$ reasoning, including the pseudocode for all of the algorithms, optimizations and evaluation. Chapter 6 describes the algorithms, optimizations and evaluation results of an Apache Spark based distributed reasoner named SparkEL. Chapter 7 contains the comparison of all the four approaches to distributed $\mathcal{EL}^+$ reasoning including the similarities and differences between the approaches. Chapter 8 contains the literature survey on distributed reasoning in various Semantic Web languages such as RDFS, OWL Horst, OWL 2 EL and OWL 2 RL. It also contains a brief description of various distributed databases. Chapter 9 summarizes the dissertation and Chapter 10 contains pointers to future work.
2 Background

2.1 Description Logics

Description Logics [55, 7] is a family of knowledge representation languages with formal semantics. They are a fragment of first order logic and they provide the formal underpinnings for OWL. The description logics that are of interest to us are $\mathcal{EL}^+$ and $\mathcal{EL}^{++}$ since they are the basis for OWL 2 EL profile.

$\mathcal{EL}^+$ [9] description logic consists of two mutually disjoint sets of atomic concept names $N_C$ and atomic role names $N_R$. $N_C^+$ is $N_C \cup \{ \top \}$. Concepts are formed according to the grammar

$$C ::= A \mid \top \mid C \cap D \mid \exists r.C,$$

where $A \in N_C^+$, $r \in N_R$, and $C, D$ over (possibly complex) concepts. An $\mathcal{EL}^+$ ontology is a finite set of general concept inclusions (GCIs) $C \sqsubseteq D$ and role inclusions (RIs) $r_1 \circ \cdots \circ r_n \sqsubseteq r$, where $C, D$ are concepts, $n$ is a positive integer and $r, r_1, \ldots, r_n$ are role names. GCIs and RIs together form the TBox of an $\mathcal{EL}^+$ ontology. The semantics of $\mathcal{EL}^+$ is provided in [9].

One of the primary reasoning tasks is called classification – the computation of the complete subsumption hierarchy between all concepts in the ontology. Other tasks, such as concept satisfiability are reducible to classification.

An example of $\mathcal{EL}^+$ axioms is given in Figure 2.1 from [8]. All the names that start
with uppercase are concepts and the ones that start with lowercase are roles. After classification of the ontology in Figure 2.1, we can get Pericarditis is a type of HeartDisease and thus needs treatment. From axiom 2.2, Pericarditis implies Inflammation which in turn implies Disease. This yields the first conjunct in the definition of HeartDisease (axiom 2.4). Since Pericardium implies ∃contained-in.Heart, ∃has-location.Pericardium implies ∃has-location.∃contained-in.Heart. Due to the role inclusion axiom 2.6, this becomes ∃has-location.Heart which is the second conjunct in the definition of HeartDisease.

\[
\begin{align*}
\text{Pericardium} & \sqsubseteq \text{Tissue} \sqcap \exists \text{contained-in.Heart} \quad (2.1) \\
\text{Pericarditis} & \sqsubseteq \text{Inflammation} \sqcap \exists \text{has-location.Pericardium} \quad (2.2) \\
\text{Inflammation} & \sqsubseteq \text{Disease} \sqcap \exists \text{acts-on.Tissue} \quad (2.3) \\
\text{HeartDisease} & \equiv \text{Disease} \sqcap \exists \text{has-location.Heart} \quad (2.4) \\
\text{HeartDisease} & \sqsubseteq \exists \text{has-state.NeedsTreatment} \quad (2.5) \\
\text{has-location} \circ \text{contained-in} & \sqsubseteq \text{has-location} \quad (2.6)
\end{align*}
\]

Figure 2.1: An example $\mathcal{EL}^+$ ontology

$\mathcal{EL}^+$ [6] description logic (which could also be called $\mathcal{ELRO}$ following more standard nomenclature [4]) is the description logic underlying the OWL 2 EL profile. $\mathcal{EL}^+$ is essentially $\mathcal{EL}^+$ extended with a bottom concept, $\bot$, and nominals $\{a\}$, where $\{a\}$ is the class containing only individual $a$. The addition is significant; e.g., it immediately allows ABox reasoning. $\mathcal{EL}^+$ also allows what are called concrete domains in concept descriptions. However, since we are interested here in the purely logical fragment of $\mathcal{EL}^+$, we omit a discussion of them. In $\mathcal{EL}^+$, nominals are included in $N_C$ and $N_C^\bot$ is defined as $N_C^T \cup \{\bot\}$.

$\mathcal{EL}^+$ appeared on the scene a few years ago [6], and received widespread attention due to the fact that it was the first known polynomial time description logic which found
application in a commercial setting: SNOMED CT ontology\(^1\) with about 300,000 axioms, falls within \(\mathcal{EL}^++\), and with the advent of the algorithm it was for the first time possible to formally classify this ontology\(^2\). This success spawned considerable and still expanding interest in polynomial-time description logics.

### 2.1.1 Ontology Normalization

Before the classification algorithm can be run on the ontology, all the axioms need to be in a **normal form**. If all the axioms in an ontology are in one of the following forms, then the ontology is said to be normalized. Concept inclusions are of the form

\[ A \sqsubseteq B | A_1 \sqsubseteq \exists r. A_2 | A_1 \cap A_2 \sqsubseteq B | \exists r.A \sqsubseteq B \]

and all role inclusions have the form \(r \sqsubseteq s\) or \(r \circ s \sqsubseteq t\). Each \(A, A_1, A_2\) and \(B\) must be in \(N^T_C\). Each \(D\) must be in \(N^C_D\).

The transformation into normal form can be done in linear time\(^3\), and the process potentially introduces concept names not found in the original ontology. The normalized ontology is a **conservative extension** of the original, in the sense that every model of the original can be extended into one for the normalized ontology.

Another step that is generally part of normalization is **internalization**, which is conversion of ABox axioms into TBox. This is especially the case if the reasoner supports only TBox axioms, i.e., the reasoner implemented only the TBox rules. Each axiom of the form \(C(a)\) is replaced by an axiom \(\{a\} \sqsubseteq C\) and each axiom of the form \(r(a, b)\) is replaced by \(\{a\} \sqsubseteq \exists r.\{b\}\). \(C(a)\) indicates that the individual \(a\) is of type \(C\) and \(\{a\} \sqsubseteq C\) indicates that a singleton class, i.e., a class with only one individual \(\{a\}\) is subclass of \(C\). Axiom \(r(a, b)\) indicates that the individuals \(a\) and \(b\) are related by the role \(r\). Singleton classes are used in the replaced axiom. In the four distributed reasoners discussed in this dissertation,

\(^1\)http://www.ihtsdo.org/snomed-ct/

\(^2\)This \(\mathcal{EL}^++\) notation is used for the extended description logic with addition of the \(\exists r\) construct.

\(^3\)The transformation can be done in linear time using a linear-time algorithm for concept subsumption.

\(^4\)\(\mathcal{EL}^++\) is an extension of \(\mathcal{EL}^+\) with the addition of the \(\exists r\) construct.

\(N^T_C\) and \(N^C_D\) refer to normal forms of TBox and ABox axioms, respectively.
internalization is part of the ontology normalization. Internalization is required since all of
the four distributed reasoners implement only TBox related rules.

In the rest of the document, we assume that all of the ontologies we deal with are
already in normal form.

2.1.2 Completion Rules

In order to classify an $\mathcal{EL}^{++}$ ontology, completion rules in Table 2.1 are used. The comple-
tion rules makes use of two mappings $S$ and $R$, where $S(X)$ maps each element $X \in N_C^\top$ to
a subset of $N_C^\bot$, and $R(r)$ maps each role name $r$ to a binary relation over $N_C^\bot$. Intuitively,
$B \in S(A)$ implies $A \sqsubseteq B$, while $(A, B) \in R(r)$ implies $A \sqsubseteq \exists r.B$. In the algorithm,
for each element $X \in N_C^\top$, $S(X)$ is initialized to contain just $\{X, \top\}$, and $R(r)$, for each
role name $r$, is initialized to $\emptyset$. The sets $S(X)$ and $R(r)$ are then extended by applying the
completion rules shown in Table 2.1 until no rule is applicable.

In Table 2.1, the condition $A \rightsquigarrow R B$ in rule R6 is used to indicate that there are
concepts $A_1, \ldots, A_k \in N_C^\top$ such that

1. $A_1 \in \{A, \top\} \cup \{\{b\}| b$ an individual$\}$,
2. $(A_j, A_{j+1}) \in R(r_j)$ for some role name $r_j$ (1 $\leq j < k$), and
3. $A_k = B$.

This condition differs slightly from the one presented in [6], which is incorrect.\footnote{The algorithm in [6] omits the case where $A_1 = \top$. It is clear, however, that this case must also be
considered.} We follow
the corrected version from [54].

The classification algorithm is guaranteed to terminate in polynomial time relative to
the size of the input ontology, and it is also sound and complete: For all class names $A$ and
$B$, $A \sqsubseteq B$ if and only if either $S(A) \cap \{B, \bot\} \neq \emptyset$, or else there is an $\{a\} \in N_C^\top$ such that
$\bot \in S(\{a\})$.\footnote{The algorithm in [6] omits the case where $A_1 = \top$. It is clear, however, that this case must also be
considered.}
Table 2.1: The classification completion rules for $\mathcal{EL}$. Rules $\textbf{R7} – \textbf{R9}$ in [6] deal with concrete domains and so are not included.

$\mathcal{EL}^+$ completion rules are a subset of $\mathcal{EL}^{++}$ completion rules and they are given in Table 2.2.

Table 2.2: The classification completion rules for $\mathcal{EL}^+$.
2.2 Distributed Computing Models

In this section, we briefly discuss the distributed computing models that are used in the rest of the document.

2.2.1 MapReduce

MapReduce is a programming model for distributed processing of data on clusters of machines (each machine being called a node) [22]. MapReduce transforms lists of input data elements into lists of output data elements. This happens twice, once in a map and again in reduce. The terms map and reduce are taken from several list processing languages such as LISP, Scheme, ML.

**Map:** The data set to be processed is divided into multiple chunks, and each chunk is assigned to a map. Map nodes generate intermediate output according to a user-defined function. In its general form, the function accepts a key-value pair and returns a set of key-value pairs. The output pairs are typically written to the local disk. The functionality of Map nodes can be represented as

\[
\text{Map} : (k_1, v_1) \mapsto \text{list}(k_2, v_2).
\]

**Reduce:** Reduce nodes are notified of the locations of intermediate output. They group values by key, and then process the values according to a user-defined Reduce function. One or more output values is produced. The general process can be represented as

\[
\text{Reduce} : (k_2, \text{list}(v_2)) \mapsto \text{list}(v_3).
\]
Map and Reduce functions are shown in Figure 2.2. Apache Hadoop (http://hadoop.apache.org) is a prominent implementation of the MapReduce model. It is an open-source framework for distributed processing of large datasets on a cluster of machines. It also provides reliable data storage facility in the form the Hadoop Distributed File System (HDFS). Developers need only define the Map and Reduce functions. Lower level and administrative tasks, such as allocating data to nodes and recovering from failures, are handled by general purpose components of the system.

An example of word count, where the occurrence of each word in a document is counted using map and reduce is given in Algorithms 1 and 2. map function receives a line as input and the first step is to tokenize, i.e., get the words in the line. For each word, map outputs the word as the key and integer 1 as the value. This indicates that it has seen a word and that word occurred once (so far). Among the key-value pairs that have been put out by the map, all the same keys end up at the same reduce. So reduce has a word as the key and a list of values (rather, a list of 1s) as value. Each reducer simply adds up the list of values and this gives the count of the occurrence of each word.
Algorithm 1: map for word count

```java
/* key: line number (not relevant), value: line */
map(key, value)

    /* split the line into words using space as the delimiter */
    wordTokens := tokenize(line);

    foreach word in wordTokens do
        emit(word, 1);
    end
```

Algorithm 2: reduce for word count

```java
/* key: word, values: list of 1s */
reduce(key, values)

    wordCount := 0;
    foreach v in values do
        wordCount := wordCount + v;
    end
    emit(wordCount);
```

Hadoop’s MapReduce is the basis of the approach described in Chapter 3.

### 2.2.2 Peer-to-Peer Computing

A distributed network architecture is called Peer-to-Peer (P2P) if resources such as computing power, storage etc. are decentralized and each node can act as both a server as well as a client [85]. All the nodes in the network collaborate with each other to accomplish a common goal. There is no hierarchical organization or centralized control among the machines in a P2P network. This is different from a Client-Server architecture where servers provide the service and clients have to request the servers to get their job done. Here client is the master since it initiates the requests and a server is the slave since it accepts the request and provides service.

P2P networks can be categorized into structured and unstructured [61]. Structured P2P network topology is tightly controlled and content is placed at specific locations so as to make the retrieval of content efficient. Structured P2P networks operate in the form of a
Distributed Hash Table (DHTs) where the storage and retrieval operations take place over data in the form of key-value pairs. The location information of the value is determined based on the key which should be unique across the cluster of machines. The primary operations that should be supported by such a network are put(key, value) and get(key). These operations involve routing requests to the peer corresponding to the key. Both the distributed reasoners, DQuEL and DistEL, discussed in chapters 4 and 5 respectively are structured P2P networks.

In unstructured P2P networks, peers do not have any prior knowledge of the topology and they can join and leave the network at any time. Data retrieval takes place by flooding the peers with the data request. When a peer receives the query, it sends the data matching the query to the originating peer. Although highly replicated data is easy to locate and retrieve, unstructured P2P networks are not suitable for efficient retrieval of rare data items. This approach is not scalable and peers become overloaded with query requests even though a particular peer might not be holding the data of interest. On the other hand, a peer in a structured P2P network can become overloaded if it holds a hot data item, i.e., data that is accessed frequently.

### 2.2.3 Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs) [108] is an abstraction that allows distributed in-memory computations over a cluster of machines in a fault-tolerant manner. RDDs are implemented in Apache Spark (http://spark.apache.org/), a general purpose framework for cluster computing. The distributed reasoning approach discussed in chapter 6 is implemented using Apache Spark. Nodes in the Spark cluster are categorized into driver and worker. Usually, there is one driver and several worker nodes. The driver initiates the application execution and the actual execution takes places in the worker nodes. This is shown in Figure 2.3. Driver hosts the SparkContext which is used to create RDDs and represents the connection to the cluster. A cluster manager allocates resources for the
tasks to be run on the workers. Executor runs on the worker node and it is responsible for running a given task and for storing the data.

![Diagram](http://spark.apache.org/docs/latest/cluster-overview.html)

Figure 2.3: Driver and worker nodes in Apache Spark. Redrawn image from [http://spark.apache.org/docs/latest/cluster-overview.html](http://spark.apache.org/docs/latest/cluster-overview.html)

RDD is a parallel data structure that is immutable and allows users to run a set of operators on the data partitions. Two types of operations are supported by RDDs: transformations and actions. A transformation is a lazy operation that takes an RDD as input, performs some computation over it and returns a new RDD. An action, on the other hand is not a lazy operation and it returns a value to the driver after running some computation on the given RDD. Infact, computations on the transformation do not get triggered unless it is followed by an action. `map` is an example of a transformation that takes an RDD and a user defined function as input. It applies the function on each element of the RDD and returns a new RDD representing the results. `count` is an action which returns the number of elements in an RDD. An example of word count using Apache Spark is given in Algorithm 3. After the data is loaded into the memory, each line of the data is split into words using space as the delimiter. For each word, integer 1 is assigned to it and for the same word, all such 1s are added. Here, `loadFile`, `flatMap` and `map` are transformations, i.e., lazy operations whereas `reduceByKey` is the only action. This `reduceByKey` action triggers the preceding transformations.
Algorithm 3: Word count in Apache Spark

```java
/* Load the data into memory */
textFile := loadFile("data.txt");
/* Each line is split into words based on the space delimiter. Each word is assigned 1 and such counts for the same word are added */
wordCounts = textFile.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey((a, b) => a + b);
```

A central theme to the design of RDDs is the lineage information that it keeps track of, which in turn is the basis of fault-tolerance. An RDD keeps track of the information of how it was derived from other RDDs through transformation operators. This metadata is represented in the form of a lineage graph (also called operator graph) with pointers to the parents of an RDD. If an RDD is lost in case of node failure, the lineage graph is used to rebuild the lost RDD. This feature lets Spark avoid expensive fault-tolerance mechanisms such as replicating data and logging updates across machines. Lineage graph is also used to build a directed acyclic graph (DAG) of the stages to execute (logical execution plan).

### 2.3 Message Passing

Ontology classification using a cluster of machines requires processes running on different machines to cooperate with each other. Two processes are said to cooperate with each other when one can affect or is affected by the other process. In order to cooperate with each other, processes need to communicate with each other. Processes that reside either on the same machine or on different machines can communicate with each other using a variety of interprocess communication mechanisms such as shared memory and message passing [88, 2]. Interprocess communication is also helpful in achieving synchronization among the processes.

Message passing is suitable for communication between processes that are residing
on two different machines. A message passing facility provides at least two operations: `send()` and `receive()`. In particular, if a process P is sending a message to process Q then the `send` and `receive` operations will be in the following form – `send(Q, message)` and `receive(P, message)`. In chapter 5, we use the symbol `!` for `send()` operation and `?` for `receive()` operation. Some of the issues associated with message passing are discussed below.

1. **Reliability of messages**

   Messages sent by processes running on different machines may not arrive or could be garbled due to the network. There are techniques to increase the reliability of data transfer such as the use of extra space in each message to put in information about the message being sent.

2. **Order of messages**

   The order in which the sender sends the messages need not be the same order in which the receiver receives them. This could be due to network delays. Some extra effort is needed, such as the use of sequence numbers or timestamps in order to ensure that the messages are delivered in the order in which they are sent.

3. **Synchronous vs Asynchronous communication**

   Interprocess communication can be either synchronous or asynchronous. In synchronous communication, the message passing primitives `send` and `receive` are blocked whereas in asynchronous communication, they are nonblocking operations. The following four possibilities exist.

   - **Blocking send** – sender is blocked until the receiver receives the message.
   - **Nonblocking send** – sender sends the message and resumes its operation. It does not wait until the receiver receives the message.
   - **Blocking receive** – receiver blocks until a message is available.
• Nonblocking receive – receiver retrieves either a valid message or null.

4. **Buffering of messages**

Messages sent by the sender are kept in a temporary queue. The size of this queue could be zero or of some finite size (theoretically, queue size could be infinite, but this is not practical). In case of a zero length queue, sender has to block until the receiver receives the message. If there is a finite size buffer available, there is a possibility for the sender to continue execution without waiting, in case there is space available in the buffer.

These issues are taken care of by the underlying message passing utility but we need to be aware of how these issues are handled since they would affect the application. In frameworks such as Hadoop and Spark, message passing is taken care of whereas in the approaches discussed in chapters 4 and 5, message passing is implemented using libraries and user-defined functions.

### 2.4 Key-Value Stores

A key-value store is a database that manages a dictionary, which is a collection of records uniquely identified by a key. They are part of the NoSQL database family [18]. *get* and *put* are the core operations supported by a key-value store. *get* retrieves data associated with the given key and *put* associates some data with the given key. Although key-value stores cannot be used to store complex data, due to its simplicity, data storage and retrieval are very fast. It is also easy to scale the data across several nodes in the cluster. A key-value store should support the following features in order to be used for distributed ontology reasoning.

i) **Good read and write speed.** All the reasoning rules from Table 2.1 involve both read and write operations. There are databases that support good read speed and has
slow writes or vice-versa. In our case, we need a database that supports good read and write speeds.

ii) **Built-in set operations.** All the reasoning rules perform set operations such as union, intersection, and membership. If the database does not support set operations, it becomes inefficient for the client to fetch the data and do the set operations on client side.

iii) **Transaction support.** In a distributed setup, a process sometimes needs to keep track of global state. Since other processes also depend on the global state, changes to it are performed either completely or not at all. So the database of interest should support transactions. There is also a possibility that several requests can be sent to the same database by different processes. Atomicity of some operations is also required.

iv) **Server-side scripting.** Operations on large amount of data is efficient if done at the server-side rather than fetching the data to the client side, perform the required operations and return the result to the server.

v) **Batch processing.** Network round-trip time on each message/request can be avoided if several operations can be batched together. Batch processing of requests generally leads to significant speed up.

vi) **Sharding.** It is a technique for partitioning a database horizontally across multiple databases usually spread across multiple machines [105]. Sharding is useful for spreading the load and improving the scalability of the system.

vii) **Good documentation and community support.** These are required in order to know the features of the database and also to use them effectively.

viii) **Support for Java.** Most of the libraries in the Semantic Web such as the OWL API[^1] and reasoner interfaces are in Java. In order to integrate well with rest of the

[^1]: [http://owlapi.sourceforge.net/]
components in the distributed reasoner, it is important that the database also supports a Java client.

There are several popular key-value stores such as Redis\footnote{http://redis.io/} [17], Dynamo\footnote{http://dynamo.yammer.com/} [23], Riak KV\footnote{http://riak.IO/} [50], HyperDEX\footnote{http://hyperdx.net/} [27], RocksDB\footnote{http://rocksdb.org/}, Aerospike\footnote{http://www.aerospike.com/}, Bigtable\footnote{http://www.google.com/bigtable/} [19] etc. We chose Redis for our distributed ontology reasoning approaches discussed in Chapters 4 and 5. Redis supports all the discussed features. It is an in-memory key-value store with excellent read and write speed. It is a single threaded server that supports set operations, sharding and Lua\footnote{http://www.lua.org} for server-side scripting. Redis also supports batch processing, referred to as pipelining. Apart from basic set operations, Redis also supports several other data structures such as lists, sorted sets, blocking lists and publish/subscribe messaging. These features are used extensively to determine the termination condition and synchronization of processes in the distributed reasoner. There is also extensive documentation on Redis and according to the database rankings (http://db-engines.com/en/ranking), it is the most popular key-value store. There is good community support in the form of mailing lists, forums and chat channels.

### 2.5 Datasets

Biomedical ontologies are used in our experiments to test the scalability of all the four distributed reasoners discussed in chapters 3, 4, 5 and 6. They vary in size, with the smallest one having only 20 axioms and the largest one having around 17 million axioms. Apart from the biomedical ontologies, DistEL (chapter 5) makes use of Traffic ontologies that have around 7 million axioms. The ontologies used, along with the information of their availability is given in Table 2.3.
### 2.5.1 Ontology Complexity

All the axioms in an $\mathcal{EL}^+$ ontology can be categorized into the following six forms where $A$, $B$, $A_1$, $A_2$ are concepts and $r$, $s$, $t$ are roles.

\begin{align*}
A \sqsubseteq B & \quad & A_1 \sqcap A_2 \sqsubseteq B \\
A \sqsubseteq \exists r.B & \quad & r.A \sqsubseteq B \\
r \sqsubseteq s & \quad & r \circ s \sqsubseteq t
\end{align*}

The completion rules (Table 2.2), described in section 2.1.2, are applied on axioms
belonging to one of the six forms. Each rule works on axiom of only one particular form
and each rule application time is different from the other. The reasoning time depends on
the following two aspects of an ontology, which we refer to as the **ontology complexity**.

1. The total number of axioms in an ontology. If the number of axioms are large in
   number, the reasoning time increases. In fact, reasoning time is polynomial in the
   size of the ontology.

2. Some completion rules are more complex than the other, i.e., the time it takes to
   process the rules varies. Rules R2, R4 and R6 involve at least one more operation
   when compared to the other three rules. Taking the analogy of relational database
   querying, if rules R1, R3 and R5 involve one join then rules R2, R4 and R6 involve
   two joins (chapter 6 involves rule implementation using joins). This makes the rules
   R2, R4 and R6 slower than the other three. Corresponding to these three rules, if
   there are more number of axioms of the form \( A_1 \sqcap A_2 \sqsubseteq B \)
   \( r.A \sqsubseteq B \) or \( r \circ s \sqsubseteq t \)
   then it slows down the reasoning time.

   Apart from these aspects of the ontology, there are other factors that influence the rea-
   soning time such as the ontology partitioning strategy and the implementation framework’s
   data storage policy. Ontology partitioning determines the amount of data that is available
   locally in order to fire a rule and the data storage policy determines whether data needs to
   be reloaded at the beginning of each iteration (reasoning is an iterative algorithm). These
   aspects are discussed in more detail in chapter 7.
Algorithm 4: replicateOntology(ontologyURI, n)

/* Load the ontology into memory */
ontology := loadOntology(ontologyURI);
ontologyConcepts := ontology.getAllConcepts();
ontologyProperties := ontology.getAllProperties();
replicatedOntology := {};

for $i = 1$ to $n$

foreach concept in ontologyConcepts do
    /* rename the concept name by appending the iteration number */
    newConcept := renameEntity(concept, concept$\_i$);
    /* replace the old concept with the renamed concept in all the axioms */
    renamedAxioms1 := renameEntityInAllAxioms(concept, newConcept);
    /* add the axioms with the renamed concept to the new ontology */
    replicatedOntology := replicatedOntology + renamedAxioms1;
end

foreach property in ontologyProperties do
    /* rename the property name by appending the iteration number */
    newProperty := renameEntity(property, property$\_i$);
    /* replace the old property with the renamed property in all the axioms */
    renamedAxioms2 := renameEntityInAllAxioms(property, newProperty);
    /* add the axioms with the renamed property to the new ontology */
    replicatedOntology := replicatedOntology + renamedAxioms2;
end

end

/* add the remaining axioms not effected by the renaming of concepts and properties to the new ontology */
replicatedOntology := replicatedOntology + remainingUnchangedAxioms;
MapReduce is a popular distributed computing framework that scales to large amount of data [58]. It is simple and easy to use since the framework takes care of all the distributed computing aspects such as inter-node communication, fault-tolerance, task management etc. The application functionality has to be represented in the form of map and reduce operations. Another reason to choose MapReduce for distributed EL+ reasoning is that it has been used to successfully scale reasoning in RDFS and OWL Horst (see Chapter 8). Although all these three profiles are Semantic Web languages, the reasoning ruleset of EL+ is different from the other two and cannot be parallelized embarrassingly. So it is more challenging to scale reasoning over large EL+ ontologies.

In this chapter, we describe the application of MapReduce for the classification of EL+ [76] and EL++ [63] ontologies. To use MapReduce with EL++, we follow the lead of [101], which describes a MapReduce algorithm for computing RDF Schema closures. However, since the completion rules from Table 2.1 are structurally more complicated than the RDF Schema completion rules, we cannot straightforwardly adopt their approach. In [100], the authors extend their approach to OWL Horst [97], facing structurally similar problems. However, due to the specific knowledge bases they are looking at, they choose a solution which is not applicable in our case.

Certain rules of the original EL++ classification algorithm are already amenable to implementation in a MapReduce framework, in the sense that the preconditions of the rules possess a common element that can serve as key. As an example of how MapReduce
could be used, consider rule R1 in Table 2.1. Ignoring the constraint ensuring that no duplicates are added to $S(X)$, the rule has only two preconditions: $A \in S(X)$ and $A \sqsubseteq B \in O$. Both of these make use of a common element $A$, and this can be used as a key. When implemented in the MapReduce framework, the map function is used to identify the preconditions relevant to R1 (indexing them using the key), and the reduce function processes the identified elements, completing the application of the rule. In the case of R1, this means that each relevant $S(X)$ is updated appropriately. Rules R2, R4, R6, and R11, however, cannot be used directly, and each must first be split into multiple rules of a more suitable form.

In the revised algorithm, new structures are used in addition to $R$ and $S$. Specifically, the functions $P$ and $L$ map each element of $N^\bot_C$ to a subset of $N^\bot_C \times N^\bot_C$, while $H$ maps each element of $N^\top_C$ to a subset of $N^\bot_C$. $N$ is a subset of $N^\bot_C$, while $J$ is a subset of $N^\bot_C \times N^\bot_C$. Intuitively, $(A_1, B) \in P(A_2)$ means $A_1 \sqcap A_2 \sqsubseteq B$. $L$ is used as a counterpart to $R$, the difference being that $L$ represents axioms with the existential restriction on the left-hand side: $(A, B) \in L(r)$ implies $\exists r.A \sqsubseteq B$. The set $J$ is used to indicate that some nominal is a superclass of two concepts: $(C, D) \in J$ implies $\{o\} \in S(C)$ and $\{o\} \in S(D)$ for some nominal $\{o\}$. $D \in N$ indicates that a sequence of the form

$A_1 \sqsubseteq \exists r_1.A_2, A_2 \sqsubseteq \exists r_2.A_3, \ldots, A_{k-1} \sqsubseteq \exists r_{k-1}.B$

holds, where $A_1$ is either $\top$ or else some nominal $\{o\}$. $B \in H(C)$ implies that a similar sequence holds, with $A_1 = C$.

The reformulation of the algorithm also requires that $R$ be altered to map binary role chains $r \circ s$ (in addition to simple roles) to a subset of $N^\bot_C \times N^\bot_C$. The intuition remains the same, however: $(C, D) \in R(r \circ s)$ implies $C \sqsubseteq \exists (r \circ s).D$. The latter expression is not grammatically correct in $EL^{++}$, but it is semantically unproblematic, and furthermore it causes no problems in the algorithm.
During the initialization of the algorithm, for each $C \in N^+_C$, $S(C)$ is set to \{\$C, \top\}$, while $P(C)$ and $H(C)$ are set to $\emptyset$. Similarly, $J, N$ are initialized to $\emptyset$, as is each $R(r)$, $R(r \circ s)$ and $L(r)$, where $r$ and $s$ are simple role names. Applying the completion rules adds to these sets.

The completion rules of the revised algorithm are shown in Table 3.1. With the exception of the removal of the constraint preventing duplicate additions (see the below discussion), rules R1, R3, and R5 are left unchanged. Rule R10 is altered to apply to both simple roles and binary role chains. This modification is semantically sound, and it does not affect the correctness of the algorithm.

<table>
<thead>
<tr>
<th>Completion Rule</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 If $A \in S(X)$ and $A \subseteq B \in \mathcal{O}$ then $S(X) := S(X) \cup {B}$</td>
<td>$A$</td>
</tr>
<tr>
<td>R2-1 If $A_1 \in S(X)$ and $A_1 \cap A_2 \subseteq B \in \mathcal{O}$ then $P(X) := P(X) \cup {(A_2, B)}$</td>
<td>$A_1$</td>
</tr>
<tr>
<td>R2-2 If $A_2 \in S(X)$ and $(A_2, B) \in P(X)$ then $S(X) := S(X) \cup {B}$</td>
<td>$A_2$</td>
</tr>
<tr>
<td>R3 If $A \in S(X), A \subseteq \exists r. B \in \mathcal{O}$ then $R(r) := R(r) \cup {(X, B)}$</td>
<td>$A$</td>
</tr>
<tr>
<td>R4-1 If $A \in S(X)$ and $\exists r. A \subseteq B \in \mathcal{O}$ then $L(r) := L(r) \cup {(X, B)}$</td>
<td>$A$</td>
</tr>
<tr>
<td>R4-2 If $(X, Y) \in R(r)$ and $(Y, B) \in L(r)$ then $S(X) := S(X) \cup {B}$</td>
<td>$r$</td>
</tr>
<tr>
<td>R5 If $(X, Y) \in R(r)$, $\bot \in S(Y)$ then $S(X) := S(X) \cup {\bot}$</td>
<td>$Y$</td>
</tr>
<tr>
<td>R6-1 If ${a} \in S(A)$ and ${a} \in S(B)$ then $J := J \cup {(A, B)}$</td>
<td>$a$</td>
</tr>
<tr>
<td>R6-2-1 If $(A, B) \in R(r)$ then $H(A) := H(A) \cup {B}$</td>
<td>$A$</td>
</tr>
<tr>
<td>R6-2-2 If $(A, B) \in R(r)$ and $A \in H(E)$ then $H(E) := H(E) \cup {B}$</td>
<td>$A$</td>
</tr>
<tr>
<td>R6-2-3 If $(A, B) \in J$ and $B \in H(A)$ then $S(A) := S(A) \cup S(B)$</td>
<td>$A$</td>
</tr>
<tr>
<td>R6-3-1 If $t \subseteq {A, B} \in R(r)$ or $(\top, B) \in R(r)$ then $N := N \cup {B}$</td>
<td>$B$</td>
</tr>
<tr>
<td>R6-3-2 If $(A, B) \in R(r)$ and $A \in N$ then $N := N \cup {B}$</td>
<td>$A$</td>
</tr>
<tr>
<td>R6-3-3 If $(A, B) \in J$ and $B \in N$ then $S(A) := S(A) \cup S(B)$</td>
<td>$B$</td>
</tr>
<tr>
<td>R10 If $(X, Y) \in R(r)$, $r \subseteq s \in \mathcal{O}$ then $R(s) := R(s) \cup {(X, Y)}$</td>
<td>$r$</td>
</tr>
<tr>
<td>R11-1 If $(X, Y) \in R(r_1)$, and $(Y, Z) \in R(r_2)$ then $R(r_1 \circ r_2) := R(r_1 \circ r_2) \cup {(X, Z)}$</td>
<td>$Y$</td>
</tr>
</tbody>
</table>

Table 3.1: Revised $\mathcal{E}^+$ algorithm. In R10, $r$ is allowed to be of the form $r_1 \circ r_2$. The keys are used in the MapReduce algorithm presented in Section 3.1.

The completion rules of the revised algorithm are shown in Table 3.1. With the exception of the removal of the constraint preventing duplicate additions (see the below discussion), rules R1, R3, and R5 are left unchanged. Rule R10 is altered to apply to both simple roles and binary role chains. This modification is semantically sound, and it does not affect the correctness of the algorithm.

The changes to the other rules are more substantial. The action of R2 is simulated by applying R2-1 and then R2-2. Rule R4 is simulated by subsequent applications of R4-1 and R4-2, and rule R11 is simulated by R11-1 and R10. E.g., if $(X, Y) \in R(r)$ and $(Y, Z) \in R(s)$, then R11-1 yields $(X, Z) \in R(r \circ s)$. If $r \circ s \subseteq t$ is an axiom of $\mathcal{O}$, then since it matches the template for R10, it follows that $(X, Z) \in R(t)$.
The rules replacing R6 are more difficult to understand. Rules R6-2-1 and R6-2-2, and also R6-3-1 and R6-3-2, are used to explicitly generate the sets $A_1, A_2, \ldots, A_k$ establishing $A \rightsquigarrow_R B$. Each pair of rules covers a distinct way in which $A \rightsquigarrow_R B$ may be established.

Specifically, R6-2-1 and R6-2-2 cover the case where $A_1 = A$. The sequence $A_1, A_2, \ldots, A_k$ is begun using rule R6-2-1 and extended with repeated applications of R6-2-2. The presence of $A_i$ in $H(A)$ implies that $(A, A_2) \in R(r), (A_2, A_3) \in R(r_2), \ldots, (A_{i-1}, A_i) \in R(r_{i-1})$ all hold, and this is sufficient to establish $A \rightsquigarrow_R A_i$.

As a concrete example, consider the following set of GCIs:

\[
\begin{align*}
A & \sqsubseteq \exists r_1.B \\
B & \sqsubseteq \exists r_2.C \\
C & \sqsubseteq \exists r_3.D \\
D & \sqsubseteq \exists r_4.E
\end{align*}
\]

Given the way $S(X)$ is initialized, it must be that $X \in S(X)$ for each $X \in N_C^\top$. As this is so, Rule R3 yields $(A, B) \in R(r_1), (B, C) \in R(r_2), (C, D) \in R(r_3)$, and $(D, E) \in R(r_4)$. Given this, it is clear that $A \rightsquigarrow_R E$ holds (according to the original definition). Using the revised rules, R6-2-1 yields $B \in H(A)$. Iterative applications of R6-2-2 yield $\{B, C, D, E\} \subseteq H(A)$, and so for any element $X \in H(A)$ (including $E$), $A \rightsquigarrow_R X$ holds.

In a similar way, rules R6-3-1 and R6-3-2 are used to cover the case where $A \rightsquigarrow_R B$ holds and the initial concept $A_1$ in the sequence $A_1, A_2, \ldots, A_k$ is a nominal $\{b\}$ or $\top$. As with $H(A)$, for each element $X \in N, A \rightsquigarrow_R X$ holds. Observe that the particular nature of $A_1$ in this case is unimportant (and it need not be related to $A$). This explains why $N$ is a set and not a function from $N_C^\top$ to $N_C^\bot$.

In rule R6-1, $(A, B) \in J$ is used solely to encode that $\{a\} \in S(A) \cap S(B)$ holds for some individual $a$. $(A, B) \in J$ is used together with the results of R6-2-1 and R6-2-2 (alternatively, R6-3-1 and R6-3-2) as input into R6-2-3 (alternatively, R6-3-3). Applying R6-2-3 or R6-3-3 completes the simulation of R6 in the original algorithm.
Each addition to $S(X)$, $P(X)$, $R(r)$, $L(A)$, $J$ and $N$ is entailed by the knowledge base $O$, and so the rules are sound. The additions also do not cause any problems with respect to termination, since there is a finite upper bound on the number of additions to any of the sets. The revised algorithm terminates if no application of any of the rules causes further additions. This alteration from the original scheme was made because it makes the algorithm more compatible with the MapReduce format. It is easy to see that the new termination condition is equivalent to the additional constraints used in the original rules. Indeed, using a straightforward inductive argument, it is easy to show that the revised algorithm yields the same results as the original, and so it is sound, complete, and terminating. It is also of polynomial worst-case complexity in the size of the input ontology; this can be shown along the lines of argument presented in [5]. The proofs for soundness, completeness and termination using some of these rules is shown in the Appendix.

3.1 Parallelization using MapReduce

The rules are now in a form amenable to implementation using the MapReduce framework. In the discussion below, we will slightly abuse terminology by referring to all expressions of the form $D \in S(C)$ (and $(C, D) \in P(E)$, $(C, D) \in R(r)$, $C \in N$, etc.), in addition to the original elements of $O$, as axioms.

The general strategy of the algorithm is as follows: The completion rules are applied in an iterative manner, with one rule being applied in a given iteration and the results of previous iterations being reused. Newly generated output is added to a database storing the contents of $O$, $S$, $P$, $R$, $H$, $N$, and $J$. Within a given iteration, the axioms obtained thus far are divided into multiple chunks. Each chunk is distributed to different computing nodes, which first act as map nodes and then as reduce nodes. Each map-reduce cycle results in the parallel application of one of the completion rules. In the map phase, based on the rule chosen to be applied, the axioms satisfying any of the preconditions of the rule are found,
and intermediate \((key, value)\) pairs are generated. In each pair, \(key\) is a concept or relation common to each precondition of the rule (as indicated in Table 3.1), while \(value\) is the axiom itself (which matches the precondition). In the reduce phase, all axioms belonging to the same key are collected from different nodes and the conclusions of the completion rule are computed, taking all valid combinations of axioms into account. All outputs are stored in a database without duplication. Iterations continue until a fixpoint is reached.

We refrain from giving descriptions of the map and reduce functions for all of the revised completion rules. MapReduce functions for rules R2-1 and R2-2 are given in Figures 5 and 6. The input of the map function is an axiom, taken from either the ontology \(O\), or else one generated from the sets \(S\), \(P\), or \(R\). Key-value pairs are generated, which are used in the reduce phase. The reduce function accepts a key and a list of values. Every possible combination of values is examined to determine whether R1-1 is applicable. A list of axioms is produced. The MapReduce functions for R6-3-2 and R6-3-3 are given in Figures 7 and 8. We will walk through an example using R6-3-3. The other rules are handled in an analogous manner, using the keys listed in Table 3.1.

\[
\begin{align*}
\{b\} & \subseteq \exists r_1.C_2 & C & \subseteq \{a\} \\
C_2 & \subseteq \exists r_2.C_3 & D & \subseteq \{a\} \\
C_3 & \subseteq \exists r_3.D & F & \subseteq \{a\} \\
D & \subseteq E & D & \subseteq G
\end{align*}
\]

Given the above inclusion axioms, rule R1 can be used to yield the following: \(\{a\} \in S(C)\), \(\{a\} \in S(D)\), \(\{a\} \in S(F)\), \(E \in S(D)\), and \(G \in S(D)\). R3 yields \((\{b\}, C_2) \in R(r_1)\), \((C_2, C_3) \in R(r_2)\), and \((C_3, D) \in R(r_3)\). R6-1 yields the following pairs in \(J\): \((C, D)\), \((C, F)\), \((D, F)\), \((D, C)\), \((F, C)\), \((F, D)\).\(^1\) Applying R6-3-1 and R6-3-2 repeatedly yields \(C_2 \in N\), \(C_3 \in N\), and \(D \in N\). Given all of these as input, R6-3-3 can now be applied. In the

\(^1\)In R6-1, we assume no pair \((C, D)\) is added to \(J\) when \(C = D\).
Algorithm 5: MapReduce algorithm for R2-1

/* key: line number (not relevant), value: an axiom */
map (key, value)
  if value == A_1 \in S(X) then
    emit((A_1, A_1 \in S(X)))
  end
  else if value == A_1 \cap A_2 \subseteq B then
    emit((A_1, A_1 \cap A_2 \subseteq B))
  end
/* key: concept, values: axioms corresponding to a rule precondition */
reduce (key, values)
  forall the v_1 in values do
    forall the v_2 in values do
      if v_1 == A_1 \in S(X) and v_2 == A_1 \cap A_2 \subseteq B then
        emit((A_2, B) \in P(X))
      end
    end
  end

map phase, the following intermediate key-value pairs are produced.

\langle C_2, C_2 \in N \rangle \quad \langle C_3, C_3 \in N \rangle \quad \langle D, D \in N \rangle
\langle D, (C, D) \in J \rangle \quad \langle C, (D, C) \in J \rangle \quad \langle F, (C, F) \in J \rangle
\langle C, (F, C) \in J \rangle \quad \langle F, (D, F) \in J \rangle \quad \langle D, (F, D) \in J \rangle

Observe that D \in N, (C, D) \in J, and (F, D) \in J are all indexed by the same key (D), and so will be processed by the same reduce node. Every combination of values passed to a reduce node will be tried. When this occurs, (C, D) \in J and D \in N cause D_i \in S(C) to be generated for each D_i \in S(D). Similarly, (F, D) \in J and D \in N cause D_i \in S(F) to be generated for each D_i \in S(D). In particular, since E \in S(D) and G \in S(D), it follows that E \in S(C), G \in S(C), E \in S(F), and G \in S(F) are all generated. Note that the values taken from S(D) and added to S(C) and S(F) are stored in a database and not included in
Algorithm 6: MapReduce algorithm for R2-2

```plaintext
/* key: line number (not relevant), value: an axiom */
map(key, value)
| if value == A_2 \in S(X) then
  | emit((A_2, A_2 \in S(X)))
end
else if value == (A_2, B) \in P(X) then
  | emit((A_2, (A_2, B) \in P(X)))
end
else if value == A_2 \sqsubseteq B then
  | emit((A_2, A_2 \sqsubseteq B))
end

/* key: concept, values: axioms corresponding to a rule precondition */
reduce(key, values)
forall the v_1 in values do
  forall the v_2 in values do
    if v_1 == A_2 \in S(X) then
      if v_2 == (A_2, B) \in P(X) or v_2 == A_2 \sqsubseteq B then
        emit(B \in S(X))
      end
    end
end
```

the values passed to the reduce node.

3.2 Theoretical Analysis

Parallelizing $\mathcal{EL}^{++}$ reasoning algorithms is hard due to the inter dependencies among the rules. Furthermore, it is known that the worst-case complexity of P-complete problems (such as $\mathcal{EL}^{++}$ ontology classification) cannot gain from parallelization [71]. Nevertheless, the ability to be able to run several computations in parallel is bound to have a significant impact on performance, at least for suitable knowledge bases. At the same time, however, this impact cannot be more than a decrease in the runtime by $k$, where $k$ is the number of
Algorithm 7: MapReduce algorithm for R6-3-2. The input of the map function is an axiom, taken from O, S, P, R, H, J, or N. Key-value pairs are generated in the map phase, and these serve as input in the reduce phase. For a given key, every possible combination of values is examined to determine whether R6-3-2 is applicable. Ultimately, a list of axioms is produced as output.

```java
/* key: line number (not relevant), value: an axiom */
map(key, value)
    if value == (A, B) ∈ R(r) then
        emit((A, (A, B) ∈ R(r)))
    end
    else if value == A ∈ N then
        emit((A, A ∈ N))
    end
/* key: a concept name or nominal; values: axioms */
reduce(key, values)
    forall the v_1 in values do
        forall the v_2 in values do
            if v_1 == (A, B) ∈ R(r) and v_2 == A ∈ N then
                emit(B ∈ N)
            end
        end
    end
```

nodes employed in parallel.

In this section, we show that in the worst case we indeed gain nothing at all. In the best case, however, we gain optimal speed-up. This indicates that the gain from parallelization depends substantially on the structure of the knowledge base. This insight is encouraging since it suggests that application scenarios for parallelization of $\mathcal{EL}^{++}$ knowledge base classification exist.

In the following, we use $n$ to indicate the size of the input knowledge base, and $k$ to indicate the number of nodes for parallelization. For our qualitative discussion, it is useful to assume that $k$ is of the order of magnitude of $n$, or to simply assume $n \leq k$, as it yields an order of magnitude for the impact of the parallelization.

For the best-case scenario, consider a knowledge base consisting of $n$ axioms $A_i \subseteq A_{i+1}$, where $1 \leq i < n$. For the complete classification, there are $n^2 - n - 1$ new inclusions
/* key: line number (not relevant), value: an axiom */

map

\[
\begin{align*}
\text{if } \text{value} \Rightarrow (A, B) & \in J \\
\text{emit}((B, (A, B) \in J)) & \\
\text{end} \\
\text{else if } \text{value} \Rightarrow A & \in N \\
\text{emit}((A, A \in N)) & \\
\text{end}
\end{align*}
\]

/* key: a concept name or nominal; values: axioms */

reduce

\[
\begin{align*}
\text{forall the } v_1 \text{ in values do} & \\
\text{forall the } v_2 \text{ in values do} & \\
\text{if } v_1 \Rightarrow (A, B) \in J \text{ and } v_2 \Rightarrow B \in N & \\
\text{forall the } E \in S(B) & \\
\text{emit}((E \in S(A)) & \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\end{align*}
\]

to be computed, which requires \(n^2 - n - 1 \in O(n^2)\) executions of rule R1. At the same
time, however the same computation can be achieved by making only \(n^2 \in O(n)\) calls to the
parallel MapReduce algorithm for the R1 rule. Note that this holds even if \(k = 1\), which
is misleading, since each MapReduce call would in this case be much more expensive
(order of magnitude: \(n\)-times as expensive) than a call to the sequential R1 rule. However,
assuming \(k \geq n\) as discussed above, the speed-up is indeed in the order of magnitude of
decreasing a quadratic time to a linear time (assuming \(k\) grows with \(n\)). Note that this is
an extreme case which potentially allows for massive parallelization through MapReduce –
and in this case having \(n\) nodes really pays off, although the assumption is unrealistic for
large \(n\). Realistic cases will usually not be as extreme.

On the other end of the spectrum is the worst-case scenario, which does not allow any
significant speed-up. Consider the knowledge base consisting of the axioms \(A_i \subseteq \exists r. A_{i+1}\),
for \(1 \leq i < n\), and \(A_n \subseteq \bot\). If we assume that the initialization of the \(R\) function according
to rule R3 has already been performed before the actual running of the algorithm (which is not an unreasonable assumption), the sequential algorithm requires $n - 1$ calls of rule R5 to arrive at the classification (namely, $A_i \sqsubseteq \bot$ for all $i$). The parallel algorithm, however, also requires $n - 1$ calls to the MapReduce version of R5, since each call results in only one new axiom.

We have ignored the general overhead which MapReduce implementations generally have. In Hadoop, for example, there is a significant overhead at start-up, which indicates that an overall gain will only be made if the input knowledge base is very large. We still must perform experiments to determine what size and what type of knowledge base is most suitable for our approach, but indications are that we will require knowledge bases which are of a size not currently available, unless artificially created. However, realistic knowledge bases will be required for a final verdict on the usability of any reasoning approach.

### 3.3 Evaluation

In order to check the performance of this approach, we implemented the ruleset of $\mathcal{EL}^+$ which is a subset of $\mathcal{EL}^{++}$ rules. This has been implemented by two research groups separately [111, 112], including our group. The experiments were run on a Hadoop cluster with 8 nodes. Each node has a 2-core, 3GHz processor with 2GB RAM. The performance of this approach and comparison with other reasoners is given in Table 3.2. GALEN ontology is replicated several times using the procedure listed in Algorithm 4 to test the scalability of this approach. ELK and jCEL are specialized OWL 2 EL reasoners whereas Pellet is a more general purpose reasoner and can work on ontologies outside of OWL 2 EL profile. This shows in the reasoning runtime of ELK and jCEL, which are much faster than Pellet.

Single machine shared memory reasoners such as ELK, jCEL and Pellet, though efficient on small ontologies, tend to run out of memory for larger ontologies. On the other hand, MapReduce approach scales with the increase in the size of the ontology. This is
Table 3.2: Classification time (in seconds) of MapReduce (MR) approach. OOM indicates Out Of Memory.

shown in Figure 3.1. With the doubling of the size of GALEN, the runtime also nearly doubles but it does not run out of memory.

Figure 3.1: Scalability of MapReduce approach on 8 nodes with different copies of GALEN ontology

3.3.1 Strengths

Following are the strengths of MapReduce based approach to $\mathcal{EL}^+$ reasoning.

1. Parallelization is built-in and the amount of parallelism can be configured using various options such as the block size of the file, number of mappers (process that runs map operation) and reducers (process that runs reduce operation).

2. Fault-tolerance is taken care of by the framework. If a node fails, other nodes are assigned the tasks that were running on the failed node.
3. In a distributed system, the total runtime of the system depends on the slowest task. A task running on a node could slow down due to a variety of hardware and software factors. Hadoop uses speculative execution to launch the slow task on other nodes. When one copy of this task finishes, Hadoop kills the other copies.

3.3.2 Runtime Inefficiency

Although this approach is scalable, it is inefficient in terms of runtime. Following are the reasons for the runtime inefficiency.

1. In the completion rules of Table 2.1 and 2.2, duplicate consequences (additions to $S$ and $R$ sets) are eliminated because of the usage of sets. Since sets are not a built-in data structure in Hadoop’s MapReduce, duplicates are generated. In order to eliminate these duplicates, an additional MapReduce job should be run. This is inefficient at two levels – a) rules are triggered on axioms that have been already considered earlier and b) additional time should be spent for duplicate removal.

2. MapReduce is best suited to tasks in which data can be divided into pieces (called chunks) and computation can be run on the chunks in parallel, i.e., it is ideal for embarrassingly parallel applications. In the case of $\mathcal{EL}^+$ reasoning, there are several dependencies among the rules, which results in communication overhead. This is not an embarrassingly parallel application.

3. In each iteration, the algorithm needs to consider only the newly generated data (compared to last iteration), so as to avoid generating duplicates. In the approach that we have taken here, it is difficult to detect and filter axioms that generate redundant inferences.

4. In every iteration, axioms are reassigned to the machines in the cluster. In the case of RDFS reasoning, schema triples are loaded in-memory and this assignment of
schema triples to machines takes place only once. This is not possible in the case of $\mathcal{EL}^+$ reasoning using MapReduce. After the end of each iteration, intermediate results are written to the disk and at the beginning of each iteration, axioms are read from the disk. This slows the runtime since disk read and write operations are expensive.
4 Distributed Queue Approach

The MapReduce based approach to $\mathcal{EL}^+$ reasoning is scalable to a limited extent and is inefficient. So we look for an alternate approach that is scalable as well as efficient. Compared to the fixpoint iteration method of rule application, it is claimed that the queue based approach is efficient on a single machine [9]. Since the queue based approach works well on a single machine, it could be possible that it would also work well in a distributed environment. In this section, we describe a distributed implementation of the queue approach and verify whether the claim also holds true in a distributed setting. First we briefly explain the queue approach on a single machine from [9] and then describe the distributed implementation of it.

4.1 Queue Algorithm

For each concept in $N_C^T$, a queue is assigned. Instead of applying the rules mechanically, in the queue approach, appropriate rules are triggered based on the type of entries in the queue. The possible entries in the queue are of the form $B_1, \ldots, B_n \rightarrow B'$ and $\exists r.B$ with $B_1, \ldots, B', B \in N_C^T$ and $r \in N_R$. If $n = 0$, $B_1, \ldots, B_n \rightarrow B'$ is simply written as $B'$. $\hat{O}$ is a mapping from a concept to sets of queue entries as follows.

- if $A_1 \cap \ldots \cap A_n \subseteq B \in \mathcal{O}$ and $A_i = A$, then $A_1 \cap \ldots \cap A_{i-1} \cap A_{i+1} \cap \ldots \cap A_n \rightarrow B \in \hat{O}(A)$
• if $A \sqsubseteq \exists r. B \in \mathcal{O}$, then $\exists r. B \in \hat{\mathcal{O}}(A)$

• if $\exists r. A \sqsubseteq B \in \mathcal{O}$, then $B \in \hat{\mathcal{O}}(\exists r. A)$

For each concept $A \in N^\top$, queue($A$) is initialized to $\hat{\mathcal{O}}(A) \cup \hat{\mathcal{O}}(\top)$. For each queue, an entry is fetched and Algorithm 9 is applied. The procedure in Algorithm 10 is called by process($A, X$) whenever a new pair of $(A, B)$ is added to $R(r)$. Note that, for any concept $A$, $\hat{\mathcal{O}}(A)$ does not change during the application of the two procedures (process, process-new-edge); $S(A)$, queue($A$) and $R(r)$ keep changing.

Algorithm 9: process($A, X$)

\begin{verbatim}
if $X = B_1, \ldots, B_n \rightarrow B'$ and $B' \notin S(A)$ then
  if $B_1, \ldots, B_n \in S(A)$ then
    continue with $X \leftarrow B'$;
  else
    return;
end
if $X$ is a concept name and $X \notin S(A)$ then
  $S(A) \leftarrow S(A) \cup \{X\}$;
  queue($A$) $\leftarrow$ queue($A$) $\cup$ $\hat{\mathcal{O}}(X)$;
  forall the concept names $B$ and role names $r$ with $(B, A) \in R(r)$ do
    queue($B$) $\leftarrow$ queue($B$) $\cup$ $\hat{\mathcal{O}}(\exists r. X)$;
end
if $X$ is an existential restriction $\exists r. B$ and $(A, B) \notin R(r)$ then
  process-new-edge($A, r, B$);
end
\end{verbatim}

4.2 Distributed Queue Approach

In the distributed setup, axioms are represented as key-value pairs as shown in Table 4.1. For axioms of the form $A_1 \sqcap \ldots \sqcap A_n \sqsubseteq B$, for each $A_i$ (key) in the conjunct,
Algorithm 10: process-new-edge($A, r, B$)

forall the role names $s$ with $r \sqsubseteq s$ do
  $R(s) \leftarrow R(s) \cup \{(A, B)\}$;
  queue($A$) \leftarrow queue($A$) \cup \bigcup_{B \mid B \in S(B)} \widehat{O}(\exists s.B');

forall the concept names $A'$ and role names $t, u$ do
  if $t \circ s \sqsubseteq u \in \mathcal{O}$ and $(A', A) \in R(t)$ and $(A', B) \notin R(u)$ do
    process-new-edge($A', u, B$);
  end
end

forall the concept names $B'$ and role names $t, u$ do
  if $s \circ t \sqsubseteq u \in \mathcal{O}$ and $(B, B') \in R(t)$ and $(A, B') \notin R(u)$ do
    process-new-edge($A, u, B'$);
  end
end

$(A_1, \ldots, A_{i-1}, A_i, \ldots, A_n, B)$ is associated as its value.

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \sqsubseteq B$</td>
<td>$A$</td>
<td>$B$</td>
</tr>
<tr>
<td>$A_1 \sqcap \ldots \sqcap A_n \sqsubseteq B$</td>
<td>$A_i$</td>
<td>$(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n, B)$</td>
</tr>
<tr>
<td>$A \sqsubseteq \exists r.B$</td>
<td>$A$</td>
<td>$(r, B)$</td>
</tr>
<tr>
<td>$\exists r.A \sqsubseteq B$</td>
<td>$(r, A)$</td>
<td>$B$</td>
</tr>
<tr>
<td>$r \sqsubseteq s$</td>
<td>$r$</td>
<td>$s$</td>
</tr>
<tr>
<td>$r \circ s \sqsubseteq t$</td>
<td>$s$</td>
<td>$(r, t)$</td>
</tr>
</tbody>
</table>

Table 4.1: Key-Value pairs for axioms

Axioms are distributed across the machines in the cluster based on their keys. A hash function, $H$ maps a unique key, $K$, to a particular node, $N$, in the cluster.

$$H : K \mapsto N$$

For each concept $A$, care is taken to map $\widehat{O}(A)$, queue($A$) and $S(A)$ to the same node. This localizes the interaction (read/write) between these three sets, which in turn improves the performance. In order not to mix up the keys among these three sets, unique namespace is used along with the key. For example, $O : A, Q : A, S : A$, for $\widehat{O}(A)$, queue($A$) and $S(A)$ respectively, but, for the hash function, $A$ is used in all the three cases.
After the axioms are loaded, each machine applies Algorithm 9 to only the queues local to it. In order to read/write to the non-local values, each machine uses the hash function, \( H \). Each machine acts as a reasoner and cooperates with other machines to get the missing values and perform the classification task.

A single process called *Termination Controller*(TC), keeps track of the status of computation across all the nodes in the cluster. TC receives either *DONE* or *NOT-DONE* message from each machine. A double check termination strategy is followed here. TC waits till it receives a *DONE* message from all the machines in the cluster. It then asks all the nodes to check if any local queues are non-empty. This is required because, after a node is done with OneIteration (Algorithm 12), there is a possibility of other nodes inserting values in the queues of this node. If this condition does indeed arise then a *NOT-DONE* message is sent to TC. TC resets its state to *NO-CHECK* and implements the double check termination strategy again. The pseudocode of TC is given in Algorithm 11. To simplify, TC is single threaded and works on only one message at a time. Process named *Job Controller* runs on each node of the cluster and implements the queue based algorithm. This is shown in Algorithm 13.

### 4.3 Evaluation

This approach is implemented in Java and the key-value store used is Redis\(^1\). Our system is called *DQuEL* and the source code is available at [https://github.com/raghavam/DQuEL](https://github.com/raghavam/DQuEL). We used a 13-node cluster with each node having two quad-core AMD Opteron 2300MHz processors and 12GB of heap size is available to JVM. Timeout limit was set to 2 hours.

\(^1\)http://redis.io

Not-GALEN, GO, NCI, SNOMED CT and SNOMEDx2 ontologies were used for
**Algorithm 11**: Termination Controller, TC

```plaintext
msgCount ← 0;
currentState ← NO-CHECK;

on pid ? status-msg do {

  if status-msg = DONE then
    msgCount ← msgCount + 1;
    if msgCount = TOTAL-NODES then
      if currentState = NO-CHECK then
        currentState ← SINGLE-CHECK-DONE;
        broadcast(CHECK-AND-RESTART);
      end
      else if currentState = SINGLE-CHECK-DONE then
        currentState ← DOUBLE-CHECK-DONE;
        broadcast(TERMINATE);
      end
    end
  else if status-msg = NOT-DONE then
    msgCount ← 0;
    currentState ← NO-CHECK;
    pid! CONTINUE-WORKING;
  end
}
```

testing. The first three are obtained from http://lat.inf.tu-dresden.de/~meng/toyont.html and SNOMED CT can be obtained from http://www.ihtsdo.org/snomed-ct. SNOMEDx2 is SNOMED replicated twice. The time taken by some popular reasoners such as Pellet, jCEL and ELK on these ontologies is given in Table 4.2. These reasoners work well on small to medium size ontologies but run out of memory on larger ontologies.

Table 4.3 shows the classification times of DQuEL with varying number of nodes. DQuEL shows good speedup on small to medium size ontologies but times out on large ontologies. Even if it is allowed to run for longer time, it runs out of memory.
Algorithm 12: OneIteration()

queues ← GetNonEmptyLocalQueues();
forall the queue $A \in$ queues do
    forall the entry $X \in A$ do
        process($A, X$);
    end
end
TC ! DONE;

Algorithm 13: Job Controller

OneIteration();
on TC ? CHECK-AND-RESTART do {
    queues ← GetNonEmptyLocalQueues();
    if queues is {} then
        TC ! DONE;
    end
    else
        TC ! NOT-DONE;
    end
} on TC ? CONTINUE-WORKING do OneIteration();
on TC ? TERMINATE do terminate-self;

4.3.1 Strengths

Following are the strengths of this approach.

1. During the ontology partitioning phase, axioms are distributed across the cluster based on the key. Good initial load balancing is possible since the hash function that maps the key to a node makes an attempt to distribute axioms across the cluster equally.

2. There is lot of flexibility if custom frameworks are used for distributed reasoning. Unlike the MapReduce approach, communication across the nodes can be established at any point and axioms can be retained in-memory after the end of each iteration.
### 4.3.2 Runtime Inefficiency

Although the results are good for smaller ontologies, this approach turns out to be inefficient for larger ontologies such as SNOMED CT. The following factors contribute to the inefficiency.

1. Batch processing of axioms is not possible because each entry in the queue could be different from the one processed before. It is a known fact that batch processing especially involving communication over networks improves the performance drastically.

2. Not all data required for the queue operations is available locally. For example, data of the form $\hat{O}(\exists r. B)$ might be present on a different node.

3. Large ontologies like SNOMED CT generate many $(X, Y)$ values for any given role key $r$ which makes computation of rules involving roles such as R4, R5 and R6 (Table 2.1) slow compared to the other rules. This also leads to data skew. The nodes that get more data are also under memory pressure. Since the data is stored in Redis,
Figure 4.1: Scalability of DQuEL on 13 nodes with different ontologies

an in-memory store, if the available RAM is not sufficient for the data, it gets written to disk. When this happens, any operation involving Redis becomes very slow and almost comes to a halt. This is the reason for the drastic decrease in the runtime of DQuEL when more nodes are added - memory pressure is relieved and Redis can operate at optimum speed.

4. The speed of the entire computation depends on the slowest task. If there are some busy nodes, ideally, the idle nodes should help the busy nodes so that the slowest task can be given more resources. But DQuEL does not support dynamic load balancing.
5 Distributed Fixpoint Iteration Approach

The MapReduce and distributed queue approach work well only on small ontologies. They do not scale well and are inefficient when reasoning over large ontologies. So an alternate approach which is scalable and shows good speed up with increase in the number of nodes is required. In this chapter, we look at the distributed version of the so called naive fixed point iteration to ontology reasoning. In fixpoint iteration approach, the completion rules are applied on the axioms iteratively until there is no new logical consequence. This idea is extended to the distributed setting [73, 75]. The distributed reasoner that was implemented based on the approach discussed in this chapter is named DistEL.

To classify an ontology, we use the completion rules given in Table 5.1 (left of the vertical line). These rules make use of three mappings $U : N_C^⊥ \rightarrow 2^{N_C^⊥}$, $R : N_R \rightarrow 2^{N_C^⊥ \times N_C^⊥}$ and $Q : N_R \rightarrow 2^{N_C^⊥ \times N_C^⊥}$ which encode certain derived consequences. More precisely, $X \in U[A]$ stands for $X \sqsubseteq A$, while $(A, B) \in R[r]$ stands for $A \sqsubseteq \exists r. B$ and $(A, B) \in Q[r]$ stands for $\exists r. A \sqsubseteq B$. For each concept $X \in N_C^⊥$, $U[X]$ is initialized to \{X, ⊥\}, and for each role $r$, $R[r]$ and $Q[r]$ are initialized to \{}. The operator $\cup= \quad$ adds elements of the set on the right-hand side to the set on the left-hand side.

The rules in Table 5.1 are applied as follows. Given a (normalized) input ontology, first initialize the $U[X]$, $R[r]$ and $Q[r]$ as indicated. Each axiom in the input knowledge base is of one of the forms given in the Table 5.1 Input column, and thus gives rise to the corresponding action given in the table. R5 and R6 are exceptions as they do not
correspond to any input axiom types, but instead they take $Q[r]$, $R[r]$ as input and trigger the corresponding action.

To compute the completion, we non-deterministically and iteratively execute all actions corresponding to all of the rules. We do this to exhaustion, i.e., until none of the actions resulting from any of the axioms causes any change to any of the $U[X]$, $R[r]$ or $Q[r]$. This is shown in Algorithm 14. Since there are only finitely many concept names, role names, and individuals occurring in the input knowledge base, the computation will indeed terminate at some stage.

The rules in Table 5.1 are from [6], except for rules R4 and R5, which is combined into one rule in [6]. Using two rules instead of one helps in the division and distribution of work in our reasoner; conceptually, we only have to store intermediate results (using $Q$, and this is the only use of $Q$ we make), and otherwise there is no difference. We also use the function $U$ instead of a function $S$ which is used in [6], where $A \in S[X]$ is used to stand for $X \sqsubseteq A$. The difference is really notational only. Our rules (and corresponding algorithm) are really just a minor syntactic variation of the original rules, and the original correctness

<table>
<thead>
<tr>
<th>Rn</th>
<th>Input</th>
<th>Action</th>
<th>Key: Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>$A \sqsubseteq B$</td>
<td>$U[B] \cup U[A]$</td>
<td>$A_{R1} : B$</td>
</tr>
<tr>
<td>R2</td>
<td>$A_1 \sqcap \cdots \sqcap A_n \sqsubseteq B$</td>
<td>$U[B] \cup U[A_1] \cap \cdots \cap U[A_n]$</td>
<td>$(A_1, \ldots, A_n)_{R2} : B$</td>
</tr>
<tr>
<td>R3</td>
<td>$A \sqsubseteq \exists r.B$</td>
<td>$R[r] \cup= {(X, B) \mid X \in U[A]}$</td>
<td>$A_{R3} : (B, r)$</td>
</tr>
<tr>
<td>R4</td>
<td>$\exists r.A \sqsubseteq B$</td>
<td>$Q[r] \cup= {(Y, B) \mid Y \in U[A]}$</td>
<td>$A_{R4} : (B, r)$</td>
</tr>
<tr>
<td>R5</td>
<td>$R[r], Q[r]$</td>
<td>$U[B] \cup= {X \mid (X,Y) \in R[r]$ and $(Y,B) \in Q[r]}$</td>
<td>$\langle$none$\rangle$</td>
</tr>
<tr>
<td>R6</td>
<td>$R[r]$</td>
<td>$U[\bot] \cup= {X \mid (X,Y) \in R[r]$ and $B \in U[\bot]}$</td>
<td>$\langle$none$\rangle$</td>
</tr>
<tr>
<td>R7</td>
<td>$r \sqsubseteq s$</td>
<td>$R[s] \cup= R[r]$</td>
<td>$r_{R7} : s$</td>
</tr>
<tr>
<td>R8</td>
<td>$r \circ s \sqsubseteq t$</td>
<td>$R[t] \cup= {(X,Z) \mid (X,Y) \in R[r]$ and $(Y,Z) \in R[s]}$</td>
<td>$r_{R8a} : (s,t)$</td>
</tr>
</tbody>
</table>

Table 5.1: Completion Rules and Key Value Pairs
Algorithm 14: General fixpoint iteration strategy for applying rules

\[
\begin{align*}
U[X] &\leftarrow \{X, \perp\}, \text{ for each } X \in N_C^\top \\
R[r] &\leftarrow \{\}, \text{ for each } r \in N_R \\
Q[X] &\leftarrow \{\}, \text{ for each } X \in N_C^\top \\
\text{repeat} & \\
& \quad \text{Old.} U[X] \leftarrow U[X]; \\
& \quad \text{Old.} R[r] \leftarrow R[r]; \\
& \quad \text{Old.} Q[X] \leftarrow Q[X]; \\
& \quad U[X] \leftarrow U[X] \cup (\text{apply } R1); \\
& \quad U[X] \leftarrow U[X] \cup (\text{apply } R2); \\
& \quad R[r] \leftarrow R[r] \cup (\text{apply } R3); \\
& \quad Q[X] \leftarrow Q[X] \cup (\text{apply } R4); \\
& \quad U[X] \leftarrow U[X] \cup (\text{apply } R5); \\
& \quad U[X] \leftarrow U[X] \cup (\text{apply } R6); \\
& \quad R[r] \leftarrow R[r] \cup (\text{apply } R7); \\
& \quad R[r] \leftarrow R[r] \cup (\text{apply } R8); \\
\text{until } & ((\text{Old.} U[X] = U[X]) \text{ and } (\text{Old.} R[r] = R[r]) \text{ and } (\text{Old.} Q[X] = Q[X]));
\end{align*}
\]

proofs carry over trivially. In section 5.2 we will comment further on the reasons we have for using \( U \) instead of \( S \): while it is only a notational variant, it is actually helpful for algorithm performance.

We use key-value pairs to encode both the input knowledge base and the output resulting from rule actions. In turn, these key-value pairs are also used to control the (then deterministic) parallel and sequential execution of rules, and this will be discussed in detail in subsequent sections.

5.1 Distributed Fixpoint Iteration Algorithms

In the algorithm descriptions in this section, we use a few CSP [2] inspired notations. The expression \( P ! \text{tag}(e) ? v \), occurring in a process \( Q \), denotes that the message \( \text{tag}(e) \) is sent to a process named \( P \) and the response received from \( P \) is assigned to \( v \). If \( P \) is not ready to receive \( \text{tag}(e) \), \( Q \) blocks until \( P \) is ready. After this message is sent, \( Q \) waits for a response from \( P \) which it will save in \( v \). \( P \) may take a while to compute this response.
But when it sends this reply, Q is ready (since it has been waiting). So P does not block when replying. The corresponding expression $Q ? \text{tag}(u)$ occurring in process P denotes receiving a message $\text{tag}(e)$ from process Q and the body of the message is assigned to variable $u$ local to P. The expression $P ! \text{tag}(e)$ occurring in a process Q simply sends a message $\text{tag}(e)$ to process P.

A process might receive many messages, and in order to distinguish between them and provide the right service to the requester, $\text{tag}$ is used. These tags are descriptive names of the service that ought to be provided.

The on statements stand for an event processing mechanism that is ever ready but asleep until triggered by a request, and the corresponding response is shown on the rhs of the do.

### 5.1.1 Partitioning the Ontology

Table 5.1 lists six unique axiom forms (excluding R5 and R6). R5 and R6 depend on the sets, $Q[r]$ and $R[r]$, for each role $r$. $Q[r]$ and $R[r]$ are set representations of axioms. For simplicity, we consider these two sets also as two separate axiom forms. This gets the total axiom forms to eight and now the input ontology $\mathcal{O}$ can be partitioned into eight mutually disjoint ontologies, $\mathcal{O} = \mathcal{O}_1 \cup \cdots \cup \mathcal{O}_8$, based on the axiom forms. Ontology $\mathcal{O}_i$ is assigned to a subcluster (subset of machines in the cluster) $SC_i$. Rule $R_i$, and no other, must be applied on $\mathcal{O}_i$. DistEL creates eight subclusters, one for each rule, from the available machines. For example (Figure 5.1) axioms that belong to $SC_4$ are divided among its three nodes. Note that, axioms in $\mathcal{O}_i$ are further divided among the machines in $SC_i$ and are not duplicated.

Assignment of nodes to each $\mathcal{O}_i/R_i$ is given in Algorithm 15. If the number of available nodes is either same as or more than the number of ontology fragments each node has to work on at least one fragment (lines 10-13). If the number of nodes are less than the ontology fragments then each node has to work on more than one ontology fragment (lines
14, 15). If the number of nodes are more than the ontology fragments, then more than one node can be assigned to work on an ontology fragment (lines 16-30). In this case, the ontology fragment is further partitioned among the nodes assigned to it. In order to do the node assignment, type and the number of axioms of each type are taken into consideration. An initial weight is assigned to each rule and each rule can be applied to only a particular type of axiom. Rules involving roles are given higher weight than other rules based on our observation that these rules take comparatively longer time. This allows more nodes to be assigned to the rules with higher weight (unless there are very few axioms of that type).

Figure 5.1: Node assignment to rules and dependency among the completion rules. A rectangle represents a node in the cluster and inner ovals represent subclusters. Outer ovals enclosing $SC_1/SC_2$, and $SC_7/SC_8$ show their tighter input-output relationships. The set (U[X], R[r], or Q[r]) affected by the rule is shown within the enclosing oval. For simplicity, only one node is shown to hold results.

Ontology partitioning should be done in such a way, so as to reduce inter-node communication. By following the described partitioning strategy, this goal is achieved since most of the data required for the rule application is available locally on each node. Other partitioning strategies such as MapReduce based data partitioning where spatial locality is
followed (data in contiguous locations are assigned to one mapper) and hash partitioning (axiom key is hashed) did not yield good results [74].

5.1.2 Rule Processes

This section presents the bodies of each of the rules of Table 5.1. These bodies are wrapped and repeatedly executed by the rule processes; this wrapper code is discussed in the termination section further below (Section 5.1.5). The service process UN is described as Algorithm 24, and RN as Algorithm 25 (Section 5.1.3). Note that, there can be any number of processes of a particular type (R1, . . . , R8, UN, RN). In all the algorithms, immediately following the forall the keywords is the retrieval of axioms, discussed further below (Section 5.1.4). Given a key such as \((Y, r)_Q\), it is fairly easy to i) extract individual values from it (such as \(Y\) and \(r\)) and ii) convert to key of different type but same values, such as \((Y, r)_{RV}\). This conversion, though not explicitly stated in all the algorithms listed here, is implicitly assumed.

Algorithms 16 and 17 follow directly from rules R1 and R2 respectively. Here (and in subsequently described algorithms), keys such as \(B_u\) correspond to those listed in Table 5.1; see also the discussion of axiom retrieval further below (Section 5.1.4). \(K_1\) (and more generally the \(K_i\) in subsequently described algorithms) are used for termination handling, as detailed towards the end of this section.

In Algorithms 18, 19 and 20, timeOf\((X)\) returns the access timestamp up to which the values of the key \(A_u\) have been read previously. Only the subsequently added values are considered. In the first on statement of Algorithm 20, the rule process R5 receives values for \((Y, r)_Q\) and \(B\) from R4. The expression \(R4 ! (Q[(Y, r)_Q] \cup \{B\})\#\) shall mean that \(\{B\}\) is added to \(Q[(Y, r)_Q]\) and that either 1 or 0 (the latter if \(B\) was already in \(Q[(Y, r)_Q]\)) is returned to R4. In the second on statement, R5 receives values for \((Y, r)_{RV}\) and \(X\) from RN. R5 gets triggered either when an axiom \(\exists r. Y \subseteq B\) is newly generated
by R4 or a new \((X, Y)\) is added to \(R[r]\), which is what these two `on` statements represent. \(\text{range}(Q[(Y, r)_Q], s, \infty)\) is a range operation on the set \(Q[(Y, r)_Q]\) in which elements starting at timestamp \(s\) and going up to the maximum available timestamp are returned. The \(Q[r]\) sets, for all roles \(r\), are maintained by rule R5 since it is the only rule process that uses them.

In Algorithm 21 for rule process R6, a set membership request is made to UN which returns a boolean value that is stored in \(b\). Algorithm 22 straightforwardly follows from rule R7.

In Algorithm 23 for rule process R8, whenever a new role pair \((X, Y)\) is added to \(R[r]\), it is checked whether this particular role \(r\) is part of any role chain axiom, say \(p \circ q \sqsubseteq t\). The two possible cases are i) \(r\) equals \(p\) or ii) \(r\) equals \(q\). Based on the case, the corresponding matching role pair is retrieved from RN.

### 5.1.3 Service Processes UN and RN

Each \(U[X]\) is a set and the process UN handles the operations over each of the \(U[X]\), for any \(X\). There can be several such UN processes which allows them to share the load. UN associates with the elements \(e\) of set \(U[X]\) a timestamp indicating when \(e\) was added to that set.

UN handles four kinds of requests, see Algorithm 24 – the first two from any arbitrary process (here named pid), the third one from R6 and the fourth from R2. The expression \((U[X] \cup= setS)\#\) stands for updating \(U[X]\) and returning the number of new items added. The first type is a request from a process named pid asking for a range of elements newly added to \(U[X]\) since its last such request made at time \(ts\). It is the responsibility of
the client to keep track of the previous timestamp up to which it has read from a particular $U[X]$. The second one is a request of the form update($X_v, D$) from pid. This updates $U$ as in $U[X] \cup= D$. Elements of $D$ are added to $U[X]$. The size increase of $U[X]$ is replied back. The third one is a membership request from R6 asking whether a particular element $Y$ is in $U[X]$. A true or false value is given as a response. The fourth is a request from R2 to retrieve the intersection of a group of $U[A_1], \ldots, U[A_n]$.

Analogous to UN, there is an RN process that handles operations over each of the $R[r]$, for any role $r$, and there can be several such RN processes sharing the load. RN handles four kinds of requests, see Algorithm 25, with most of them similar to the requests handled by UN. The time stamp $ts$ is sent in by the requester. Whenever RN receives an update message with a new role pair $((Y, r)_{RV}, X)$, it notifies the processes (R5, R6, R7, R8) that depend on $R[r]$ values. A new role pair is duplicated on the rule process R8 for further processing. This is done because it is more efficient than separate retrieval of the right role pair using a key. However, this duplication is not required in all cases: If, for a particular role $r$, this $r$ does not appear in the second position of the chain, (e.g., in the position of $q$ as in $p \circ q \sqsubseteq t$), then this particular $R[r]$ is not duplicated.

The expression $(R[(Y, r)_{RV}] \cup= \{X\})#$ stands for updating $(R[(Y, r)_{RV}]$ and returning the number of new items, zero or one, added.

### 5.1.4 Retrieval of Axioms from the Key Value Store

We use key-value stores [18] to keep the eight parts of the ontology including the $U[X]$, the $R[r]$ and the $Q[r]$, for all concepts $X$ and roles $r$. Each of these is maintained by separate service processes. The $O_i$ processes are co-located with the $R_i$ rule processes. We retrieve axioms from the $O_i$ services in the forall the ... do statements.

Concepts and roles are mnemonic strings of the ontology and we encode them as integers. E.g., 032560 represents a concept (indicated by the last 0) whose ID is 256. The
length of the ID is given in the first two positions (03 in this case).

Table 5.1 shows the keys and their corresponding values for axioms in the ontology. Axioms have a left hand side and a right hand side with respect to $\sqsubseteq$. In most cases, the left hand sides becomes the key and right hand side the value, both encoded as unsigned 64-bit integers. The paired expressions yield an integer from which the paired items can be peeled off. The hash of the concepts is used in encoding them as keys.

The choice of key is not straightforward. For example, for axioms of type $A \sqsubseteq \exists r.B$ (R3), making $r$ as the key would lead to load imbalance since there are generally only a few roles in an ontology and comparatively many axioms of type $A \sqsubseteq \exists r.B$. On the other hand, making $A$ as key leads to better load distribution, thus allowing several machines to work on $R[r]$.

R8 gets triggered when there is a change to either $R[r]$ or $R[s]$. In order to retrieve the exact match, i.e., given $(X, Y)$ of $R[r]$, get $(Y, Z)$ of $R[s]$ or vice versa, the $R[r]$ sets, for any $r$, have two keys $(Y, r)_R$ and $(X, r)_R$. The $R[r]$ sets are selectively duplicated. For the same reason, there are two keys for the role chain axioms as well.

### 5.1.5 Termination

Algorithm 26 invokes the rule process $R_i$ on the axioms in $O_i$ once i.e., $R_i$ is applied on the axioms one time and the updates made to the $U[X]$ and $R[r]$ sets are collected in $K_i$ (this could be 0). Notice that a $K_i$ is associated with each $R_i$ in Algorithms 16–23. This value is broadcast to all the other rule processes. Then it waits for similar update messages to be received from other rule processes. Barrier synchronization [2] is used in waiting for $K_i$ from all $R_i$ (indicated by the barrier-sum statement). If no rule process made an update, they quit; otherwise, they continue with another iteration. The same termination condition is used for processes handling $U[X]$ and $R[r]$ sets (Algorithm 27). Algorithms 26 and 27 act as wrappers around the other processes $R_i$, UN, RN.
This termination condition is easy to check on a single machine. But in a distributed system, termination is no longer obvious. For example, just when the process working on rule R1 is done and quits, the next moment, a process working on rule R5 might add a new \( B \) to \( U[X] \). Although barrier synchronization simplifies the termination detection, it also makes several nodes wait idly. This idleness is reduced in our system using a work stealing mechanism, which is detailed in Section 5.2.

### 5.2 Optimizations

We discuss some of the efficiency optimizations we have realized in our approach.

1. \( U[X] \) instead of \( S[X] \)

   \( S[X] \) defined as \( A \in S[X] \) iff \( X \subseteq A \) is used in the original formulation of the algorithm in [6]. We recast this as \( U[X] \) defined as \( A \in U[X] \) iff \( A \sqsubseteq X \). Use of \( U[X] \) instead of \( S[X] \) makes the check \( A \in S[X] \), which is required in several rules, a single read call, and thus significantly more efficient.

   For example, assume that there are five concepts in the ontology, \( K, L, M, N \) and \( P \). Suppose \( K \sqcap L \sqcap M \sqsubseteq N \in \mathcal{O} \). During some iteration of the classification assume \( S(K) = \{K, L, N, \top\} \), \( S(L) = \{L, P, M, \top\} \), \( S(M) = \{M, N, K, \top\} \), \( S(N) = \{N, \top\} \), and \( S(P) = \{P, K, L, M, \top\} \). Now, according to rule R2 in [6], we have to check for the presence of \( K, L \) and \( M \) in each of the five \( S(X) \), where \( X = K, L, M, N, P \). Since only \( S(P) \) has \( K, L, M \), we have to add \( N \) to \( S(P) \).

   On the other hand, we use instead \( U[K] = \{K, M, P\} \), \( U[L] = \{L, K, P\} \), \( U[M] = \{M, L, P\} \), \( U[N] = \{N, K, M, P\} \), \( U[P] = \{P, L\} \). In this case, instead of checking all \( U[X] \), we can compute the intersection of \( U[K], U[L], U[M] \), which is \{\( P \)\}. So, \( P \sqsubseteq N \) which is represented as \( U[N] \cup= \{P\} \). In large ontologies, the number of concepts could be in the millions, but the number of conjuncts in axioms like **61**
\( A_1 \cap \cdots \cap A_n \subseteq B \) would be very low. So the performance is better by using \( U[X] \) since set intersection needs to be performed only on a very small number of sets.

2. **Rule Dependencies**

Say rule R3 just finished processing axiom \( \alpha = A \sqsubseteq \exists r.B \). If none of R1, R2, R5 or R6 make any changes to \( U[A] \), R3 need not be triggered again to consider \( \alpha \). If and when R3 gets triggered again, it resumes from entries in \( U[A] \) with a later timestamp. Thus, we reduce the number of axioms to work on in subsequent iterations.

3. **Dynamic Load Balancing**

Processing time for each of the rules, R1 to R8, varies due to the number and type of axioms. This can lead to improper load balancing where there are busy and idle nodes. We apply the well known work stealing mechanism [60], where idle nodes take (steal) work from busy nodes, thus reducing their load. Although this is a well known idea, to the best of our knowledge, there is no freely available distributed work stealing library. Although work stealing increases the communication cost, performance improvement outweighs it. Algorithms for work stealing are given in the next section (Section 5.3).

### 5.3 Work Stealing

The axioms handled by each node are divided into several fixed sized pieces known as chunks. A chunk forms a basic unit of processing. A node works on only one chunk at any point of time. After processing a chunk, it picks up the next available local chunk. A faster node, after finishing processing of all its local chunks, can steal a chunk of work (axioms) from a busy node, thus reducing the work of busy node.

Pseudocode for work stealing is given in Algorithm 28. Process running on each node invoke processAxioms(). First step is to divide the work to be done into chunks which is
done in Algorithm 29. The number of axioms that form a chunk is configurable.

A barrier synchronization is required after dividing the work into chunks because there is a chance that not all processes would finish dividing work into chunks on their nodes at the same time. In the meanwhile, a node could become idle and ask for a chunk of work from another node which hasn’t yet completed the call to divideWorkIntoChunks().

After dividing the work into chunks, a node should read the next available chunk. This is shown in Algorithm 30. In order to prevent a chunk from being processed by more than one node, as soon as a chunk is read, it is deleted from the database. Hence these operations should be atomic. All these operations are performed on the node on which the process runs.

Processing a work chunk (Algorithm 31) involves applying a rule on that chunk, broadcasting the work progress message and saving the status obtained after processing the rule.

After processing all the local chunks, a node has to wait for any process which stole work from this node. This is required in order to evaluate the number of changes made across all the chunks. This is represented by the call waitForWorkStealers() in Algorithm 28. After broadcasting the status message, a node is free to steal a chunk of work from a busy node. This is described in Algorithm 32. Since there is a mapping between nodes and rules, from the machine information, rule type can be obtained. As shown in Algorithm 32, a work chunk would be stolen one after the other until all the work chunks across the cluster are processed.
5.4 Evaluation

We believe that it is possible to distribute computation of the completion of OWL EL ontologies in such a way that the distributed approach ... 

(Claim 1) scales to very large ontologies to finish the classification task and 

(Claim 2) shows reasonable speedup in the number of nodes.

We verified these claims by implementing a prototype in Java, called DistEL, downloadable from http://github.com/raghavam/DistEL. We used Redis\(^1\), a key-value store, as our database. Redis was selected because it provides excellent read/write speed along with built-in support for set operations, database sharding, transactions and server-side scripting.

Since one of the use cases is streaming traffic data, DistEL also has support for incremental classification. It is inherently supported, since, in each iteration of the classification procedure, only the newly added axioms are considered and appropriate rules are applied.

We used Amazon’s Elastic Cloud Compute (EC2) to run our experiments. Specifically, we used m3.xlarge instances which have 4 cores, 15GB RAM and SSD hard disk. 5GB was given to the JVM on each node, for all the experiments. These settings and the m3.xlarge instances were selected so as to evaluate our system on a cluster of machines with commodity hardware.

5.4.1 Test Data

Our test data (see Table 5.2) comprises of biomedical ontologies GO\(^2\), SNOMED CT\(^3\) and traffic data of the city of Dublin, Ireland.\(^4\) We also duplicated 2x, 3x and 5x copies of \(^{\text{\textsuperscript{1}http://redis.io}}\)
\(^{\text{\textsuperscript{2}http://code.google.com/p/elk-reasoner/wiki/TestOntologies}}\)
\(^{\text{\textsuperscript{3}http://www.ihtsdo.org}}\)
\(^{\text{\textsuperscript{4}Raw data of the traffic ontology is from http://dublinked.ie/datastore/datasets/dataset-215.php. This data is converted to }\mathcal{EC}^{++} \text{ ABox statements as described in [57]. The}}\)
SNOMED CT using the procedure outlined in Algorithm 4 in order to test the scalability of our system.

Traffic data reasoning is used in the diagnosis and prediction of road traffic congestions \cite{56, 57}. These tasks depend on (i) classifying any new individual from the ontology stream, and (ii) identifying their causal relationships and correlation with other streams such as city events. There is no bound on the number of axioms since it is a continuous stream of traffic data. In this scenario, existing reasoners were not able to cope with the increasing velocity and volume of data. Here, we considered traffic data of only one single day. Data is collected every 20 seconds and we have 1441 such bursts.

\subsection{5.4.2 Results}

Table 5.3 has the classification times for ELK 0.4.1, jCEL 0.19.1, Snorocket 2.4.3, Pellet 2.3.0, HermiT 1.3.8 and FaCT++ 1.6.2. All the reasoners are invoked through the OWL API and ontology loading time is excluded wherever applicable.

All the reasoners ran out of memory on the SNOMEDx3, SNOMEDx5 and Traffic. On traffic data, incremental classification has been used by the reasoners that support it (ELK, Pellet, HermiT). This experiment with single machine reasoners demonstrates that a scalable solution is required to handle large ontologies.

Table 5.4 shows the classification times of our system as we add more and more nodes. The cluster size need not be in multiples of 8. DistEL is able to classify all the ontologies including the largest one having close to 74 million axioms. This validates Claim 1 of our hypothesis.

Table 5.7 shows the speedup achieved by DistEL on SNOMED CT with increasing

\footnotesize{\begin{itemize}
    \item TBox statements (base ontology), along with two samples of ABox statements, are available from http://www.dropbox.com/sh/9jnutinqj188heu/AAAi-5ot8A5fStz69Bd0VyGCa.
\end{itemize}}
number of nodes. As can be seen, there is a steady increase in the speedup with increase in the number of nodes. This validates Claim 2 of our hypothesis. The speedup along with the scalability of DistEL can also be seen from Figures 5.2 and 5.3. Excluding GO (a small ontology), for all the other large ontologies, classification time decreases as we increase the number of nodes. On 64 nodes, we notice an increase in the runtime for all but the largest of the ontologies. This indicates that beyond a point, the advantages of the distributed approach are overshadowed by the distribution and communication overhead. However, this is not the case for largest ontology, traffic data. We believe this is due to the
axiom composition in traffic data. 75% of traffic data axioms are in the form of \( A \subseteq \exists r.B \) (R3). The output of R3 serves as input to R5, R6, R7 and R8 i.e., 63% of nodes are always busy, i.e. there are more busy nodes than idle nodes. This is not the case as such for the other ontologies.

Table 5.5 shows the memory (RAM) taken by Redis in MB on each of the 8 nodes for traffic data. In this case, only one node is used to collect the results (\( U[X] \) sets). \( R[r] \) sets are spread across other nodes. As can be seen, each node takes very little memory. But on single machine reasoners, this quickly adds up for large ontologies and current reasoners hit their limit in terms of memory (see Table 5.3) and computational power.

### 5.4.3 Discussion

We believe DistEL is the first distributed reasoner for EL ontologies that is scalable and is able to achieve good speedup. So we cannot do a like-for-like comparison with any other distributed reasoner. At the risk of being skewed, the following are our observations in comparison to ELK, which is the fastest reasoner among the ones we tested on (see Table 5.3).

Table 5.6 shows the speedup of ELK on SNOMED on an 8 core machine. For DistEL, 8 nodes was the starting point. Considering that ELK is a shared memory system with all the threads on one machine, the speedup achieved by DistEL (Table 5.7) is very reasonable in comparison. On this basis, we can say that our design and optimization decisions (Sections 5.1.1, 5.2) are justified.

DistEL on 8 nodes for SNOMED takes 544 seconds whereas ELK takes 32 seconds. Classification is not “embarrassingly parallel”, so linear speedup cannot be achieved. Since axioms are distributed across many nodes, communication is necessary. Another contributing factor is the mismatch in the speed of in-memory and Redis operations (Table 5.8). This
is a simple experiment where 1 million integers are read and written to a Java HashMap. Similar operations were performed on a Redis hash data structure.\(^5\) Although this is a rather simple experiment, the difference in read/write speeds in the case of RAM and Redis is quite obvious.

These experiments suggest that a distributed approach should be used only on very large ontologies where the size/complexity of ontologies simply overwhelms current reasoners. Thus a distributed approach has potential benefits which are quite complementary to single machine reasoners.

5.4.4 Runtime Inefficiency

DistEL shows good scalability and speedup on large ontologies especially when compared with the MapReduce approach (Chapter 3) and distributed queue approach (Chapter 4). But compared to the single machine shared memory reasoners such as ELK, the reasoning runtime is inefficient. Following are some of the reasons for this inefficiency in DistEL.

1. There are two processes running on each node of the cluster - a java process that applies the rule and a database server process (Redis) that holds the data. Both these processes are single threaded. So they do not take advantage of the multiple cores on each node.

2. Although the approach described in this chapter supports multiple result nodes, in the implementation and evaluation only one result node is used (Figure 5.1). This puts more pressure on the result node and other nodes have to wait for their request to be processed by the result node, in case the result node is busy.

3. Although Redis is an in-memory key-value store, as indicated in Table 5.8, it is slower compared to Java’s data structures. Since several database operations take

\(^5\)The code used for this experiment is available at https://gist.github.com/raghavam/2be48a98cae31c418678.
place in each iteration, this difference in speed plays a crucial role in contributing to
the runtime inefficiency.

4. Barrier synchronization is used for termination detection. All the processes have
to wait for the slowest process to complete before proceeding to the next iteration.
Although the effect of barrier synchronization is mitigated to a good extent by work
stealing, nodes still have to spend some time (at least a few seconds) waiting.

5. Some of the optimizations involve fetching data from other nodes in the cluster. If
the data fetch happens several times or if the amount of data per fetch is more then
the optimization is as bad or worse than not having the optimization. For larger
ontologies, the time spent on some of the optimizations might have a negative impact
on the reasoning runtime.
Algorithm 15: Algorithm for node assignment i.e., from among the available number of nodes, this algorithm calculated the number of nodes to be assigned to each rule/ontology fragment.

assignNodes()
| Let \( n \) be the total nodes in the cluster, \( n > 0 \).
| Let \( r \) be the total number of rules.
| Let the weight assigned to rule \( R_i \) be \( w_i, 0 < w_i \leq 1 \).
| \( t_1, t_2, \ldots, t_r \) represent the number of axioms of each type.
| \( A_1, A_2, \ldots, A_n \) represent available nodes.
| \( N_1[], N_2[], \ldots, N_r[] \) represent the list of nodes assigned to each rule.
| \( p_1, p_2, \ldots, p_r \) be the nodes that can be assigned to each rule based on score.
| \( i \leftarrow 1, j \leftarrow 1; \)
| while \( i \leq r \) do
| \( N_i[1] \leftarrow A_j; \) // assign at least one node to each rule
| \( i \leftarrow i + 1; \)
| \( j \leftarrow j + 1; \)
| if \( j > n \) then
| | \( j \leftarrow 1; \)
| end
| end
| if \( n > r \) then
| \( A_{r+1}, A_{r+2}, \ldots, A_k \) be the available nodes after initial assignment.
| \( m \leftarrow n - r; \)
| \( S \leftarrow 0; \)
| for \( i \leftarrow 1 \) to \( r \) do
| | \( s_i \leftarrow w_i \ast t_i; \) // \( s_i \) is the score of each rule
| | \( S \leftarrow S + s_i; \) // \( S \) is the total score
| end
| for \( i \leftarrow 1 \) to \( r \) do
| | \( p_i \leftarrow (s_i/S) \ast m; \)
| end
| \( D \leftarrow \text{sortInDesc}(p_1, p_2, \ldots, p_r); \) // sort them in descending order
| forall the \( d_i \in D \) do
| | if \( d_i \leq m \) then
| | | add \( d_i \) nodes to \( N_i[]; \)
| | | delete \( d_i \) nodes from available nodes;
| | | \( m \leftarrow m - d_i; \)
| | end
| end
| end
Algorithm 16: R1: \( A \subseteq B \Rightarrow U[B] \subseteq U[A] \)

\[ K_1 := x := 0; \]
\[
\text{forall the } A \subseteq B \in \mathcal{O}_1 \text{ do}
\]
\[ \text{UN} ! \text{ update}(B_u, A) ? x; \]
\[ K_1^+ = x; \]
\end

Algorithm 17: R2: \( A_1 \cap \cdots \cap A_n \subseteq B \Rightarrow U[B] \cup U[A_1] \cap \cdots \cap U[A_n] \)

\[ K_2 := x := 0; \]
\[
\text{forall the } A_1 \cap \cdots \cap A_n \subseteq B \in \mathcal{O}_2 \text{ do}
\]
\[ \text{UN} \cap (B_u, \{A_1, \ldots, A_n\}) ? x; \]
\[ K_2^+ = x; \]
\end

Algorithm 18: R3: \( A \subseteq \exists r.B \Rightarrow R[r] \cup = \{(X, B) | X \in U[A]\} \)

\[ K_3 := x := 0; \]
\[
\text{forall the } A \subseteq \exists r.B \in \mathcal{O}_3 \text{ do}
\]
\[ s := \text{timeOf}(A_v); \]
\[ \text{UN} ! \text{ queryTS}(A_u, s) ? M; \]
\[
\text{forall the } X \in M \text{ do}
\]
\[ \text{RN} ! \text{ update}((B, r)_Y, X) ? x; K_3^+ = x; \]
\end

Algorithm 19: R4: \( \exists r.A \subseteq B \Rightarrow Q[r] \cup = \{(Y, B) | Y \in U[A]\} \)

\[ K_4 := x := 0; \]
\[
\text{forall the } \exists r.A \subseteq B \in \mathcal{O}_4 \text{ do}
\]
\[ s := \text{timeOf}(A_v); \]
\[ \text{UN} ! \text{ queryTS}(A_u, s) ? M; \]
\[
\text{forall the } Y \in M \text{ do}
\]
\[ \text{R5} \text{ new}((Y, r)_Q, B) ? x; \]
\[ K_4^+ = x; \]
\end
Algorithm 20: R5: \((X, Y) \in R[r] \land (Y, B) \in Q[r] \Rightarrow U[B] \cup= \{X\}\)
\[
K_5 := x := 0;
\]
\textbf{on} R4 \? new((Y, r)_Q, B) \textbf{do}
\{
R4 \!: (Q[(Y, r)_Q] \cup= \{B\})#;
s := timeOf((Y, r)_RY);
RN \! queryTS((Y, r)_RY, s) ? T;
\textbf{forall the} X \in T \textbf{do}
  \textbf{UN} \! update(B_U, X) \? x;
  K_5 += x;
\}
\}
\textbf{on} RN \? rpair((Y, r)_RY, X) \textbf{do}
\{
s := timeOf((Y, r)_Q);
T := range(Q[(Y, r)_Q], s, \infty);
\textbf{forall the} B \in T \textbf{do}
  \textbf{UN} \! update(B_U, X) \? x;
  K_5 += x;
\}
\}

Algorithm 21: R6: \(X \subseteq \exists r.Y \Rightarrow U[\bot] \cup= \{X \mid Y \in U[\bot]\}\)
\[
K_6 := x := 0;
\]
\textbf{on} RN \? yxpair(Y_{R6}, X) \textbf{do}
\{
UN \! isMember(\bot_U, Y_{R6}) \? b;
\textbf{if} b \textbf{then}
  \textbf{UN} \! update(\bot_U, X) \? x;
  K_6 += x;
\textbf{end}
\}
\}

Algorithm 22: R7: \(r \sqsubseteq s \Rightarrow R[s] \cup= R[r]\)
\[
K_7 := x := 0;
\]
\textbf{on} RN \? rpair((Y, r)_RY, X) \textbf{do}
\textbf{forall the} s \textbf{ (with} r \sqsubseteq s \in O_7 \textbf{do}
  \textbf{RN} \! update((Y, s)_RY, X) \? x; K_7 += x;
\textbf{end}
Algorithm 23: R8: \( r \circ s \sqsubseteq t \Rightarrow R[t] \cup= \{(X, Z) \mid (X, Y) \in R[r], (Y, Z) \in R[s]\}\)

\[ K_8 := x = 0; \]

\[ \text{on RN ? rpair}((Y, r)_{RY}, X) \text{ do } \{
  \text{forall the } s, t \ (\text{with } r \circ s \sqsubseteq t \in O_8) \text{ do}
  \text{RN ! queryX}((Y, s)_{RX}) \? T;
  \text{forall the } Z \in T \text{ do}
    \text{RN ! update}((Z, t)_{RY}, X) \? x;
    K_8 += x;
\end{equation}

\[ \text{forall the } s, t \ (\text{with } s \circ r \sqsubseteq t \in O_8) \text{ do}
  \text{RN ! queryY}((X, s)_{RY}) \? T;
  \text{forall the } Z \in T \text{ do}
    \text{RN ! update}((Y, t)_{RY}, Z) \? x;
    K_8 += x;
\end{equation}

\[ \text{end} \]

\[ \text{end} \]

\[ \text{on RN ? isOnLHS2}(s_{R8b}) \text{ do } \{ b := \exists s_{R8b}; \text{RN ! b } \}; \]

Algorithm 24: Process UN maintains \( U[X] \), for all \( X \).

\[ \text{on pid ? queryTS}(U_X, \text{ts}) \text{ do } \{ T := \text{range}(U[X], \text{ts}, \infty); \text{pid ! T } \}; \]

\[ \text{on pid ? update}(U_X, \{A_1, \ldots, A_n\}) \text{ do pid ! (U[X] \cup= \{A_1, \ldots, A_n\})#}; \]

\[ \text{on R6 ? isMember}(X_U, Y) \text{ do R6 ! (Y \in U[X])}; \]

\[ \text{on R2 ?} \cap(B_U, \{A_1, \ldots, A_n\}) \text{ do R2 ! (U[B] \cup= U[A_1] \cap \cdots \cap U[A_n])#}; \]

Algorithm 25: Node RN maintains \( R[r] \) sets

\[ \text{on R5 ? queryTS}((Y, r)_{RY}, \text{ts}) \text{ do } \{ T := \text{range}(R[(Y, r)_{RY}], \text{ts}, \infty); \text{R5 ! T } \}; \]

\[ \text{on R8 ? queryX}((X, r)_{RX}) \text{ do R8 ! (R[(X, r)_{RX}])}; \]

\[ \text{on R8 ? queryY}((Y, r)_{RY}) \text{ do R8 ! (R[(Y, r)_{RY}])}; \]

\[ \text{on pid ? update}((Y, r)_{RY}, X) \text{ do } \{
  \text{pid ! (R[(Y, r)_{RY}] \cup= \{X\})#};
  \text{R5 ! rpair}((Y, r)_{RY}, X);
  \text{R6 ! yxpair}(Y_{R6}, X);
  \text{R7 ! rpair}((Y, r)_{RY}, X);
  \text{R8 ! rpair}((Y, r)_{RY}, X);
  \text{R8 ! isOnLHS2}(r_{R8b}) \? b;
  \text{if } b \text{ then}
    \text{R[(X, r)_{RX}] \cup= \{Y\}};
  \text{end}
  \end{equation} \]
Algorithm 26: Wrapper for Ri

\begin{algorithm}
\SetAlgoLined
\Repeat{\text{nUpdates} = 0}
\State \( K_i := \text{apply Ri on } O_i \) \text{ once;}
\State \text{broadcast}(K_i);
\State nUpdates := barrier-sum-of \( K_i \);
\EndRepeat
\end{algorithm}

Algorithm 27: Wrapper for UN and RN

\begin{algorithm}
\SetAlgoLined
\Repeat{\text{nUpdates} = 0}
\State nUpdates := barrier-sum-of \( K_i \);
\EndRepeat
\end{algorithm}

Algorithm 28: Algorithm to process axioms with work stealing

\begin{algorithm}
\SetAlgoLined
\text{processAxioms()}
\State \text{divideWorkIntoChunks(} localhost\text{);}  
\State /* wait until all the processes divide work into chunks on their nodes */
\State \text{barrierSynch();}
\State \text{status} \leftarrow \text{false;}
\Repeat{\text{chunkCount} = -1}
\State \{w, \text{chunkCount}\} \leftarrow \text{getAWorkChunk(} localhost\text{);}  
\If{\text{chunkCount} \neq -1}
\State \text{processOneWorkChunk(} w, localhost, \text{chunkCount}\text{);}  
\EndIf
\Until{\text{chunkCount} = -1;}
\State /* wait for all the local chunks to be processed */
\State \text{waitForWorkStealers();}
\State \text{statusMessages} \leftarrow \text{readStatusMsgsFromDB();}
\ForAll{\text{statusMsg} \in \text{statusMessages}}
\State \text{status} \leftarrow \text{status} \lor \text{statusMsg;}
\EndFor
\State \text{broadcast(} status\text{);}  
\State \text{stealWork();}
\end{algorithm}

Algorithm 29: Algorithm to divide the set of axioms into chunks

\begin{algorithm}
\SetAlgoLined
\text{divideWorkIntoChunks(} host\text{)}
\State \text{divide the work (set of axioms) } \mathcal{W} \text{ of host into chunks;}
\State \mathcal{W} \leftarrow \mathcal{W}_1 \cup \mathcal{W}_2 \cup \cdots \cup \mathcal{W}_k;
\end{algorithm}
Algorithm 30: Algorithm to retrieve a work chunk

```
getAWorkChunk(host)
    /* All the method calls here are performed on the host */
    atomic {
        c ← readChunk();
        if c ≠ null then
            chunkCount ← incrementChunkCounter();
            deleteChunk(c);
        end
        else
            chunkCount ← -1;
        end
    }
    return {c, chunkCount};
```

Algorithm 31: Algorithm to process one chunk of work

```
processOneWorkChunk(wChunk, host, chunkCount)
    /* get total chunks of work from the remote host */
    totalChunks ← getTotalChunks();
    if chunkCount ≠ -1 then
        progress ← chunkCount/totalChunks;
    end
    else
        /* work is complete */
        progress ← 1.0;
    end
    broadcast(progress);
    /* get the rule from host-rule mapping on the remote host */
    R_i ← getRule();
    /* apply the appropriate completion rule locally on the axioms obtained from the busy node */
    k ← applyCompletionRule(R_i, wChunk);
    /* k indicates the number of new logical consequences */
    saveInDB(k);
```
**Algorithm 32:** Algorithm to steal the work from the busiest node

```plaintext
stealWork()
while true do
    /* from the progress messages, determine the most busy node */
    remoteHost ← getBusiestMachineInfo();
    if (remoteHost == null) then // its null when work in all nodes is done
        return;
    end
    {w, chunkCount} ← getAWorkChunk(remoteHost);
    if chunkCount ≠ -1 then
        processOneWorkChunk(w, remoteHost, chunkCount);
    end
end
```

#### Table 5.2: Number of axioms, before and after classification, in ontologies.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>GO</th>
<th>SNOMED</th>
<th>SNOMEDx2</th>
<th>SNOMEDx3</th>
<th>SNOMEDx5</th>
<th>Traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>87,137</td>
<td>1,038,481</td>
<td>2,076,962</td>
<td>3,115,443</td>
<td>5,192,405</td>
<td>7,151,328</td>
</tr>
<tr>
<td>After</td>
<td>868,996</td>
<td>14,796,555</td>
<td>29,593,106</td>
<td>44,389,657</td>
<td>73,982,759</td>
<td>21,840,440</td>
</tr>
</tbody>
</table>

#### Table 5.3: Classification times in seconds. OOM\(^a\): reasoner runs out of memory. OOM\(^b\): reasoner runs out of memory during incremental classification. OOM\(^c\): ontology too big for OWL API to load in memory.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>ELK</th>
<th>jCEL</th>
<th>Snorocket</th>
<th>Pellet</th>
<th>Hermit</th>
<th>FaCT++</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO</td>
<td>23.5</td>
<td>57.4</td>
<td>40.3</td>
<td>231.4</td>
<td>91.7</td>
<td>367.89</td>
</tr>
<tr>
<td>SNOMED</td>
<td>31.8</td>
<td>126.6</td>
<td>52.34</td>
<td>620.46</td>
<td>1273.7</td>
<td>1350.5</td>
</tr>
<tr>
<td>SNOMEDx2</td>
<td>77.3</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
</tr>
<tr>
<td>SNOMEDx3</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
</tr>
<tr>
<td>SNOMEDx5</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
<td>OOM(^a)</td>
</tr>
<tr>
<td>Traffic</td>
<td>OOM(^b)</td>
<td>OOM(^c)</td>
<td>OOM(^b)</td>
<td>OOM(^b)</td>
<td>OOM(^b)</td>
<td>OOM(^c)</td>
</tr>
</tbody>
</table>

#### Table 5.4: Classification time (in seconds) of DistEL

<table>
<thead>
<tr>
<th>Ontology</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>24 nodes</th>
<th>32 nodes</th>
<th>64 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO</td>
<td>134.49</td>
<td>114.66</td>
<td>109.46</td>
<td>156.04</td>
<td>137.31</td>
</tr>
<tr>
<td>SNOMED</td>
<td>544.38</td>
<td>435.79</td>
<td>407.38</td>
<td>386.00</td>
<td>444.19</td>
</tr>
<tr>
<td>SNOMEDx2</td>
<td>954.17</td>
<td>750.81</td>
<td>717.41</td>
<td>673.08</td>
<td>799.07</td>
</tr>
<tr>
<td>SNOMEDx3</td>
<td>1362.88</td>
<td>1007.16</td>
<td>960.46</td>
<td>928.41</td>
<td>1051.80</td>
</tr>
<tr>
<td>SNOMEDx5</td>
<td>2182.16</td>
<td>1537.63</td>
<td>1489.34</td>
<td>1445.30</td>
<td>1799.13</td>
</tr>
<tr>
<td>Traffic</td>
<td>60004.54</td>
<td>41729.54</td>
<td>39719.84</td>
<td>38696.48</td>
<td>34200.17</td>
</tr>
</tbody>
</table>
### Table 5.5: Memory taken by Redis on each node for traffic data

<table>
<thead>
<tr>
<th>Node</th>
<th>MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>186.72</td>
</tr>
<tr>
<td>R2</td>
<td>0.81</td>
</tr>
<tr>
<td>R3</td>
<td>257.47</td>
</tr>
<tr>
<td>R4</td>
<td>0.79</td>
</tr>
<tr>
<td>R5</td>
<td>1970</td>
</tr>
<tr>
<td>R6</td>
<td>380.61</td>
</tr>
<tr>
<td>R7</td>
<td>0.79</td>
</tr>
<tr>
<td>R8</td>
<td>1470.00</td>
</tr>
<tr>
<td>Result</td>
<td>654.53</td>
</tr>
<tr>
<td>Total</td>
<td>4921.72</td>
</tr>
</tbody>
</table>

### Table 5.6: Speedup achieved by ELK, with all the threads on one 8-core machine, on SNOMED CT

<table>
<thead>
<tr>
<th>Threads</th>
<th>Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31.80</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>19.37</td>
<td>1.64</td>
</tr>
<tr>
<td>3</td>
<td>16.29</td>
<td>1.95</td>
</tr>
<tr>
<td>4</td>
<td>14.91</td>
<td>2.13</td>
</tr>
<tr>
<td>5</td>
<td>13.99</td>
<td>2.27</td>
</tr>
<tr>
<td>6</td>
<td>14.16</td>
<td>2.24</td>
</tr>
<tr>
<td>7</td>
<td>13.17</td>
<td>2.41</td>
</tr>
<tr>
<td>8</td>
<td>13.36</td>
<td>2.38</td>
</tr>
</tbody>
</table>

### Table 5.7: Speedup achieved by DistEL on SNOMED CT

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>544.38</td>
<td>1.00</td>
</tr>
<tr>
<td>16</td>
<td>435.79</td>
<td>1.24</td>
</tr>
<tr>
<td>24</td>
<td>407.38</td>
<td>1.33</td>
</tr>
<tr>
<td>32</td>
<td>386.00</td>
<td>1.41</td>
</tr>
<tr>
<td>64</td>
<td>444.19</td>
<td>1.22</td>
</tr>
</tbody>
</table>

### Table 5.8: Speed (in seconds) for simple read, write operations of 1,000,000 items using RAM and Redis

<table>
<thead>
<tr>
<th>Operation</th>
<th>RAM</th>
<th>Redis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>0.0861</td>
<td>3.719</td>
</tr>
<tr>
<td>Write</td>
<td>0.1833</td>
<td>4.688</td>
</tr>
</tbody>
</table>

Table 5.5: Memory taken by Redis on each node for traffic data

Table 5.6: Speedup achieved by ELK, with all the threads on one 8-core machine, on SNOMED CT

Table 5.7: Speedup achieved by DistEL on SNOMED CT

Table 5.8: Speed (in seconds) for simple read, write operations of 1,000,000 items using RAM and Redis
6 Spark based Approach

The distributed reasoner named DistEL discussed in the previous chapter offers good scalability and speedup on large ontologies. But the reasoning runtime is poor, especially when compared with the existing single machine shared memory reasoners. So we look for an alternate approach to distributed reasoning that addresses some of the issues (Section 5.4.4) mentioned in the previous chapter. The approach discussed in this chapter is also a fixed point iteration algorithm where the completion rules of Table 6.1 are applied on the axioms repeatedly until there is no new logical consequence. In this chapter, we describe such an approach implemented using an open source general purpose distributed computing framework called Apache Spark\(^1\). Resilient Distributed Dataset (RDD) is the primary abstraction (discussed in Section 2.2.3). This work was done in collaboration with Sambhawa Priya, a PhD student at Lehigh University.

The rules in Table 6.1 make use of \(S\) and \(R\), which were described in earlier chapters. But they are repeated here for the sake of convenience. \(S\) and \(R\) are defined as \(S : N_C^\top \rightarrow 2^{N_C^\top}\), \(R : N_R \rightarrow 2^{N_C^\top \times N_C^\top}\). More precisely, \(A \in S(X)\) stands for \(X \sqsubseteq A\), while \((A, B) \in R(r)\) stands for \(A \sqsubseteq \exists r.B\). In addition to \(S\) and \(R\), we also make use of \(U\). \(X \in U(A)\) if \(X \sqsubseteq A\), i.e., \(U\) is simply a flip of \(S\). For each concept \(X \in N_C^\top\), \(S(X)\) is initialized to \(\{X, \top\}\), and for each role \(r\), \(R(r)\) is initialized to \(\{}\). We make use of the operator \(\cup=\) in the pseudocode (Section 6.2.1). This adds elements of the set on the
If $A \in S(X)\text{, }A \sqsubseteq B \in \mathcal{O}\text{, and }B \notin S(X)$, then $S(X) := S(X) \cup \{B\}$

If $A_1, A_2 \in S(X)\text{, }A_1 \cap A_2 \sqsubseteq B \in \mathcal{O}\text{, and }B \notin S(X)$, then $S(X) := S(X) \cup \{B\}$

If $A \in S(X)\text{, }A \sqsubseteq \exists r.B \in \mathcal{O}\text{, and }(X, B) \notin R(r)$, then $R(r) := R(r) \cup \{(X, B)\}$

If $(X, Y) \in R(r)\text{, }A \in S(Y)\text{, }\exists r.A \sqsubseteq B \in \mathcal{O}\text{, and }B \notin S(X)$, then $S(X) := S(X) \cup \{B\}$

If $(X, Y) \in R(r)\text{, }r \sqsubseteq s \in \mathcal{O}\text{, and }(X, Y) \notin R(s)$, then $R(s) := R(s) \cup \{(X, Y)\}$

If $(X, Y) \in R(r_1)\text{, }(Y, Z) \in R(r_2)\text{, }r_1 \circ r_2 \sqsubseteq r_3 \in \mathcal{O}\text{, and }(X, Z) \notin R(r_3)$, then $R(r_3) := R(r_3) \cup \{(X, Z)\}$

Table 6.1: The classification completion rules for $\mathcal{EL}^+$. 

right-hand side to the set on the left-hand side.

A normalized $\mathcal{EL}^+$ ontology $\mathcal{O}$ can be partitioned into six mutually disjoint ontologies, $\mathcal{O} = \mathcal{O}_1 \cup \cdots \cup \mathcal{O}_6$, based on the axioms on which the six rules can be applied. We refer to $\mathcal{O}_1$ as type1Axioms, $\mathcal{O}_2$ as type2Axioms, $\mathcal{O}_3$ as type3Axioms, $\mathcal{O}_4$ as type4Axioms, $\mathcal{O}_5$ as type5Axioms, and $\mathcal{O}_6$ as type6Axioms. All these different types of axioms are spread across the cluster and at any point of time, only one rule is applied on the axioms of a particular type by all the machines in the cluster.

### 6.1 Axiom Representation

Axioms are represented in the form of key-value pairs as shown in Table 6.2. The usage of this format makes indexing of various axioms very easy and leads to a cleaner design of algorithms. This also helps in the implementation of algorithms in Apache Spark.

The choice of key is very important as it affects not only the distribution of axioms across the cluster but also join operations. These issues are discussed in more detail in Section 6.3.

---

1[^1](http://spark.apache.org/)
### Table 6.2: Axiom format for different axiom types

<table>
<thead>
<tr>
<th>Axiom Type</th>
<th>Axiom Format</th>
<th>Type Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \subseteq B )</td>
<td>((A, B))</td>
<td>type1Axioms</td>
</tr>
<tr>
<td>( A_1 \cap A_2 \subseteq B )</td>
<td>(((A_1, A_2), B))</td>
<td>type2Axioms</td>
</tr>
<tr>
<td>( A \subseteq \exists r. B )</td>
<td>((A, (r, B)))</td>
<td>type3Axioms</td>
</tr>
<tr>
<td>( \exists r. A \subseteq B )</td>
<td>((A, (r, B)))</td>
<td>type4Axioms</td>
</tr>
<tr>
<td>( r \subseteq s )</td>
<td>((r, s))</td>
<td>type5Axioms</td>
</tr>
<tr>
<td>( r \circ s \subseteq t )</td>
<td>((r, (s, t)))</td>
<td>type6Axioms</td>
</tr>
<tr>
<td>( S(X) = {A, B, \ldots} )</td>
<td>((X, A), (X, B), \ldots)</td>
<td>sAxioms</td>
</tr>
<tr>
<td>( U(A) = {X, Y, \ldots} )</td>
<td>((A, X), (A, Y), \ldots)</td>
<td>uAxioms</td>
</tr>
<tr>
<td>( R(r) = {(X, Y), (K, M), \ldots} )</td>
<td>((r, (X, Y)), (r, (K, M)), \ldots)</td>
<td>rAxioms</td>
</tr>
</tbody>
</table>

### 6.2 Algorithms

In this section, we describe the algorithms used in the implementation of all the six completion rules. We make use of certain operations such as map, join, filter, union and distinct.

**map** applies a function on each element of the collection on which the map is called. For example, in uAxioms := sAxioms.map((x, a) \Rightarrow (a, x)), an anonymous function \((x, a) \Rightarrow (a, x)\) is applied on each element of sAxioms collection. Left side of \(\Rightarrow\) represents the input parameter to the anonymous function which is each element of the collection and right side of \(\Rightarrow\) is the function body. This map operation swaps the key and value positions in the sAxioms collection.

**filter** applies a function on each element of the collection on which the filter is called and returns only those elements that satisfy the filter criteria. So the function that the filter applies on each element should return a boolean value. For example, in sAxiomsReflexive := sAxioms.filter((x, a) \Rightarrow x == a), all such elements where both the key and value are the same are retained.

**join** performs a natural join on the key of two collections that are in key-value format. For example, sAxiomsSelfJoin := sAxioms.join(sAxioms), is a self join on sAxioms collection. sAxioms are in the form of \((x, a)\). The join output format is \((x, (a1, a2))\), i.e., the common key in the two join operands is retained and a compound value which is a comma
separated list of the values from the two join operands is created.

union operation over two or more collections combines them into one. This operation retains duplicates. All the operands should be in the same format.

distinct operation removes duplicates from a collection.

6.2.1 Pseudocode for Completion Rules

We now describe the naive version of algorithm for all the six completion rules.

Completion Rule 1

Pseudocode for rule R1 is given in Algorithm 33. We explain this algorithm with the help of an example.

Steps for an example

1. Let \( A \subseteq B \) and \( U(A) = \{ X \} \). So type1Axioms has \((A, B)\) and uAxioms has \((A, X)\)

2. type1Axioms is joined with uAxioms to produce \((A, (B, X))\)

3. Here we are only interested in the value, so we extract the value from \((A, (B, X))\), which is \((B, X)\)

4. \((B, X)\) is the newly inferred logical consequence and should be added to the uAxioms collection. So we add to it using the union operation and remove duplicates \(((B, X)\ could\ be\ a\ duplicate)\).

Completion Rule 2

Pseudocode for rule R2 is given in Algorithm 34. We explain this algorithm with the help of an example.

Steps for an example
Algorithm 33: R1: $A \sqsubseteq B$, $A \in S(X) \Rightarrow S(X) \cup= \{B\}$

```plaintext
/* (a, b) ⊗ (a, x) ⇒ (a, (b, x)) */
r1Join := type1Axioms.join(uAxioms);
/* (a, (b, x)) ⇒ (b, x) */
r1JoinMap := r1Join.map((a, (b, x)) ⇒ (b, x)); /* union the resulting axioms with original set of axioms */
uAxiomsNew := uAxioms.union(r1JoinMap);
/* remove duplicate axioms */
uAxioms := uAxiomsNew.distinct();
```

1. Let $A_1 \cap A_2 \subseteq B$, $U(A_1) = \{X_1\}$ and $U(A_2) = \{X_2\}$. So type2Axioms has $(A_1, (A_2, B))$ and uAxioms has $((A_1, X_1), (A_2, X_2))$.

2. type2Axioms is joined with uAxioms to produce $(A_1, ((A_2, B), X_1))$

3. Since we next need to join on $A_2$, we rearrange $(A_1, ((A_2, B), X_1))$ so that $A_2$ becomes the key, i.e., $(A_2, (B, X_1))$. We no longer need $A_1$. Let us call this collection as r2Join2Input.

4. r2Join2Input joined with uAxioms produces a collection with elements in the form of $(A_2, ((B, X_1), X_2))$. We need to filter this collection to find those elements where $X_1$ and $X_2$ match, i.e., find such $U$ which have a common $X$.

5. After filtering them, value field is extracted and added to the uAxioms and duplicates are removed. The overall computation in rule R2 can be shown as (uAxioms ⊗ type2Axioms) ⊗ uAxioms ⇒ uAxiomsNew.

**Completion Rule 3**

Pseudocode for rule R3 is given in Algorithm 35. We explain this algorithm with the help of an example.

**Steps for an example**
Algorithm 34: R2: \( A_1 \cap A_2 \subseteq B, A_1, A_2 \in S(X) \Rightarrow S(X) \cup= \{ B \} \)

```java
/* (a1, (a2, b)) \bowtie (a1, x1) ⇒ (a1, ((a2, b), x1)) */
r2Join1 := type2Axioms.join(uAxioms);
/* (a1, ((a2, b), x1)) ⇒ (a2, (b, x1)) */
r2Join2Input := r2Join1.map((a1, ((a2, b), x1))) ⇒ (a2, (b, x1));
/* (a2, (b, x1)) \bowtie (a2, x2) ⇒ (a2, ((b, x1), x2)) */
r2Join2 := r2Join2Input.join(uAxioms);
/* Keep only axioms of form (a2, ((b, x), x)) */
r2Join2Map := r2Join2.filter((a2, ((b, x1), x2)) ⇒ x1 == x2);
/* union the resulting axioms with original set of axioms */
r2Output := r2Join2Map.map((a2, ((b, x1), x2)) ⇒ (b, x1));
/* union the resulting axioms with original set of axioms */
r2Output := r2Output.union(r2Output);
/* remove duplicate axioms */
uAxiomsNew := uAxioms.union(r2Output);
/* remove duplicate axioms */
uAxioms := uAxiomsNew.distinct();
```

1. Let \( A \subseteq \exists r.B \) and \( U(A) = \{ X \} \). So type3Axioms has \( (A, (r, B)) \) and uAxioms has \( (A, X) \)

2. type3Axioms joined with uAxioms produces a collection containing elements of the form \( (A, ((r, B), X)) \). We are only interested in the value part. So we extract the value part and rearrange it to produce elements of the form \( (r, (X, B)) \)

3. This result is added to rAxioms collection and duplicates are removed

Algorithm 35: R3: \( A \subseteq \exists r.B, A \in S(X) \Rightarrow R(r) \cup= \{ X, B \} \)

```java
/* (a, (r, b)) \bowtie (a, x) ⇒ (a, ((r, b), x)) */
r3Join := type3Axioms.join(uAxioms);
/* (a, ((r, b), x)) ⇒ (r, (x, b)) */
r3Output := r3Join.map((a, ((r, b), x)) ⇒ (r, (x, b)));
/* union the resulting axioms with original set of axioms */
r3Output := r3Output.union(r3Output);
/* remove duplicate axioms */
rAxiomsNew := rAxioms.union(r3Output);
/* remove duplicate axioms */
rAxioms := rAxiomsNew.distinct();
```
Completion Rule 4

Pseudocode for rule R4 is given in Algorithm 36. We explain this algorithm with the help of an example.

**Steps for an example**

1. Let $\exists r. A \sqsubseteq B$, $U(A) = \{Y\}$ and $R(r) = \{(X, Y)\}$. So $type4Axioms$ has $(A, (r, B))$, $uAxioms$ has $(A, Y)$ and $rAxioms$ has $(r, (X, Y))$

2. $type4Axioms$ joined with $uAxioms$ produces a collection with elements of the form $(A, ((r, B), Y))$. This is rearranged to $((r, Y), B))$. $A$ is no longer required for further operations, so it can be dropped. Let us call this collection $r4Join1CompoundKey$

3. $rAxioms$ are also rearranged from $(r, (X, Y))$ to $((r, Y), X)$, so that the key matches that of $r4Join1CompoundKey$ axioms. Let us call this collection $rAxiomsPairYKey$

4. $r4Join1CompoundKey$ joined with $rAxiomsPairYKey$ produces a collection where elements are in the form of $((r, Y), (B, X))$. We need only the value $(B, X)$ here. So it is extracted and added to $uAxioms$. Duplicates are removed.

**Algorithm 36: R4:** $(X, Y) \in R(r)$, $A \in S(Y)$, $\exists r. A \sqsubseteq B \Rightarrow S(X) \sqsubseteq \{B\}$

```plaintext
/* $(a, (r, b)) \bowtie (a, y) \Rightarrow (a, ((r, b), y)) */
r4Join1 := type4Axioms.join(uAxioms);
/* $(a, ((r, b), y)) \Rightarrow ((r, y), b) */

/* $(r, (x, y)) \Rightarrow ((r, y), x) */
r4Join1CompoundKey := r4Join1.map((a, ((r, b), y)) \Rightarrow ((r, y), b));

/* $(r, (x, y)) \Rightarrow ((r, y), x) */
rAxiomsPairYKey := rAxioms.map((r, (x, y)) \Rightarrow ((r, y), x));
/* $((r, y), b) \bowtie ((r, y), x) \Rightarrow ((r, y), (b, x)) */

/* $((r, y), (b, x)) \Rightarrow (b, x) */
r4Join2 := r4Join1CompoundKey.join(rAxiomsPairYKey);
/* union the results with current total $uAxioms$ */
uAxiomsNew := uAxioms.union(r4Result);
/* remove duplicates */
uAxiomsNew := uAxiomsNew.distinct();
```
Completion Rule 5

Pseudocode for rule R5 is given in Algorithm 37. We explain this algorithm with the help of an example.

Steps for an example

1. Let \( r \sqsubseteq s \) and \( R(r) = \{(X,Y)\} \). So type5Axioms has \( (r,s) \) and rAxioms has \( (r,(X,Y)) \)

2. type5Axioms joined with rAxioms produces a collection whose elements are in the form of \( (r,(s,(X,Y))) \). \( r \) is no longer needed. So we rearrange it to produce \( (s,(X,Y)) \). This is added to rAxioms and duplicates are removed.

Algorithm 37: R5: \( (X,Y) \in R(r), r \sqsubseteq s \Rightarrow R(s) \cup= \{(X,Y)\} \)

```plaintext
/* (r, s) ▷ (r, (x, y)) ⇒ (r, (s, (x, y))) */
r5Join := type5Axioms.join(rAxioms);
/* (r, (s, (x, y))) ⇒ (s, (x, y)) */
r5Result := r5Join.map((r, (s, (x, y))) ⇒ (s, (x, y)));
/* union the results with current total rAxioms */
rAxiomsNew := rAxioms.union(r5Result);
/* remove duplicates */
rAxiomsNew := rAxiomsNew.distinct();
```

Completion Rule 6

Pseudocode for rule R6 is given in Algorithm 38. We explain this algorithm with the help of an example.

Steps for an example

1. Let \( r_1 \circ r_2 \sqsubseteq r_3 \), \( R(r_1) = \{(X,Y)\} \) and \( R(r_2) = \{(Y,Z)\} \). So type6Axioms has \( (r_1,(r_2,r_3)) \) and rAxioms has \( ((r_1, (X,Y)), (r_2, (Y,Z))) \)
2. type6Axioms joined with rAxioms produces a collection whose elements are in the form of \((r_1, ((r_2, r_3), (X, Y)))\). We no longer need \(r_1\) in further operations, so we rearrange it to produce \(((r_2, y), (r_3, x))\). Let us call this collection \(r6Join1CompoundKey\).

3. \(rAxioms\) are rearranged from \(((r_2, (y, z))\) to \(((r_2, y), (r_3, x))\) so that the key matches that of \(r6Join1CompoundKey\). Let us call this collection \(rAxiomsCompoundKey\).

4. \(r6Join1CompoundKey\) is joined with \(rAxiomsCompoundKey\) to produce a collection whose elements are of the form \(((r_2, y), ((r_3, x), z))\). We no longer need \(r_2\) and \(y\). So we rearrange it to produce \((r_3, (x, z))\). This is added to \(rAxioms\) and duplicates are removed.

---

**Algorithm 38:**

R6: \((X, Y) \in R(r_1), (Y, Z) \in R(r_2), r_1 \circ r_2 \sqsubseteq r_3 \Rightarrow R(r_3) \cup= \{(X, Z)\}\)

```java
/* (r_1, (r_2, r_3)) ≃ (r_1, ((r_2, r_3), (x, y))) */
r6Join1 := type6Axioms.join(rAxioms);
/* (r_1, ((r_2, r_3), (x, y))) ⇒ ((r_2, y), (r_3, x)) */
r6Join1CompoundKey := r6Join1.map((r1, ((r2, r3), (x, y))) ⇒ ((r2, y), (r3, x)));
/* (r_2, (y, z)) ⇒ ((r_2, y), (r_3, x), z) */
rAxiomsCompoundKey := rAxioms.map((r2, (y, z)) ⇒ ((r2, y), (r3, x), z));
/* ((r_2, y), ((r_3, x), z)) ⇒ ((r_2, y), ((r_3, x), z)) */
r6Join2 := r6Join1CompoundKey.join(rAxiomsCompoundKey);
/* ((r_2, y), ((r_3, x), z)) ⇒ ((r_3, x), z) */
r6Result := r6Join2.map(((r2, y), ((r3, x), z)) ⇒ (r3, (x, z)));
/* union the results with current total rAxioms */
rAxiomsNew := rAxioms.union(r6Result);
/* remove duplicates */
rAxiomsNew := rAxiomsNew.distinct();
```

---

**Wrapper Algorithm**

Pseudocode for the wrapper algorithm is given in Algorithm 39. Each completion rule is applied on the axioms until no further axioms are generated. The termination condition is
Algorithm 39: Wrapper algorithm that applies all the rules on the axioms until completion

\[
\begin{align*}
\text{prevUAxiomsCount} &:= 0; \\
\text{prevRAxiomsCount} &:= 0; \\
\text{currUAxiomsCount} &:= \text{uAxioms.count();} \\
\text{currRAxiomsCount} &:= \text{rAxioms.count();} \\
\textbf{while} (\text{prevUAxiomsCount} \neq \text{currUAxiomsCount}) \lor (\text{prevRAxiomsCount} \neq \text{currRAxiomsCount}) \textbf{do} \\
\ R1(); &\quad \text{apply Algorithm 33} \\
\ R2(); &\quad \text{apply Algorithm 34} \\
\ R3(); &\quad \text{apply Algorithm 35} \\
\ R4(); &\quad \text{apply Algorithm 36} \\
\ R5(); &\quad \text{apply Algorithm 37} \\
\ R6(); &\quad \text{apply Algorithm 38} \\
\end{align*}
\]

\[
\begin{align*}
\text{prevUAxiomsCount} &:= \text{currUAxiomsCount}; \\
\text{prevRAxiomsCount} &:= \text{currRAxiomsCount}; \\
\text{currUAxiomsCount} &:= \text{uAxioms.count();} \\
\text{currRAxiomsCount} &:= \text{rAxioms.count();} \\
\textbf{end}
\end{align*}
\]

checked by keeping track of the uAxioms and rAxioms count. If the count does not change with respect to the previous iteration, then the reasoning process can be terminated.

6.3 Optimizations

1. **Pruning Join Operands.** When doing a natural join on two collections, we can use the more selective collection to filter the non-selective collection. A join in a distributed environment would involve moving the data across the cluster, also referred to as shuffling the data in Apache Spark. So filtering a large collection would help in the transfer of lesser data. Keys of the smaller collection can be used in filtering the keys of larger collection. For example, in the following join, \text{type4Axioms} . \text{join(uAxioms)}, type4Axioms are much smaller compared to uAxioms. So the keys of type4Axioms, i.e., \( A \) in \((A, (r, B))\) can be used to filter the uAxioms collection.
2. **Join Ordering and Key Ordering.** If there are multiple join operands, the order in which they are joined greatly influences the cost and performance of the operation, more so in a distributed environment. The selectivity of the key also plays a role in the performance of join. A compound key can have more selectivity when compared to a key with a single element in it. Rules involving multiple joins such as completion rule 2, completion rule 4 and completion rule 6 were optimized using the join ordering.

3. **Delta Computations.** In the algorithms described in Section 6.2, completion rules were applied on the entire uAxioms and rAxioms collection. This generates lots of duplicate axioms, i.e., axioms that were already inferred in previous iterations since completion rules are being applied on the same uAxioms and rAxioms that were already considered in previous iterations. Completion rules should be applied on a subset of uAxioms and rAxioms that are newly generated and haven’t been considered yet. Such axioms are referred to as \( \Delta uAxioms \) and \( \Delta rAxioms \). Note that, all uAxioms and rAxioms should also be considered along with \( \Delta uAxioms \) and \( \Delta rAxioms \) in order for the reasoning to be sound. For example, in rule R4 (Algorithm 36), both uAxioms and rAxioms are involved in the joins. If \( \Delta uAxioms \) is used for the join in \( r4Join1 := type4Axioms.join(uAxioms) \), then full rAxioms collection should be used for the join in \( r4Join2 := r4Join1CompoundKey.join(rAxiomsPairYKey) \) and vice-versa.

### 6.4 Implementation Details

Our distributed reasoner, named SparkEL, was built using a general purpose distributed computing framework called Apache Spark\(^2\) 1.6.1. SparkEL was implemented in Scala 2.10.6 and is available at [https://github.com/raghavam/sparkel](https://github.com/raghavam/sparkel). Apache Spark was chosen because i) it is good for iterative algorithms, especially the ones that

\(^2\)http://spark.apache.org/
converge with respect to a particular condition and ii) it can parallelize and optimize the computations. We used OWL API 4.1.3 to read the ontologies and invoke the reasoners listed in Table 6.4.

All the experiments were run on Microsoft Azure D3 v2 machines that have 4 cores and 14GB RAM. Network bandwidth is 1 Gbps and disk has 8KB of 500 IOPS.

### 6.4.1 Spark Framework related Optimizations

1. **Caching.** In iterative algorithms caching the right data is very important. Spark recomputes the data that has not been cached. uAxioms and rAxioms from previous iteration is used in the next iteration and new axioms are added to it. So these two collections keep growing. uAxioms and rAxioms should be cached at the end of each iteration so that it is available for the next iteration without any recomputation.

   If we keep caching data from each iteration, it will run out of space and Spark starts recomputing the data. Unused data needs to be removed from cache. The data removal strategy that we follow is that at the end of iteration $n$, uAxioms and rAxioms from iteration $n - 1$ are removed from cache.

2. **Broadcast Data.** Spark moves the data around nodes for certain operations such as join. For example, in $r6Join1 := type6Axioms.join(rAxioms)$, elements with the same keys from type6Axioms and rAxioms are moved to the same node.

   Moving the data around is an expensive operation. In order to avoid this, some data can be replicated on all the nodes. The broadcast feature of Spark allows us to cache the data on all the nodes. Generally, data that is small and does not change throughout the computation is broadcast. For example, type1Axioms, ..., type6Axioms can be broadcast since they do not change during the entire reasoning process.

3. **Hash Partitioning.** If data cannot be broadcast, we can make sure that the operands
involved in the join are present on the same node. Hash partitioning involves hashing the key and assigning the tuple (key, value) to a specific partition, which in turn is assigned to a node. Keys that are same end up on the same node. For example, consider the join from rule R6 (Algorithm 38), \( r6Join2 := r6Join1CompoundKey \) \( \text{join}(rAxiomsCompoundKey) \). Let \( r6Join1CompoundKey \) be \( ((r2, y), (r3, x)) \) and \( rAxiomsCompoundKey \) is \( ((r2, y), z) \). Since the keys are the same, they end up on the same node and join is no longer an expensive operation.

4. Partition-aware functions.

- **Distinct.** The fixpoint iteration algorithm generates lots of duplicates. So removal of duplicates is necessary. The default distinct operation supported by Spark involves moving the data (shuffle) since it hashes on both the key and value. If the data is hash partitioned, it is sufficient to check for duplicates within a partition since two values with the same key end up in the same partition. The pseudocode for this customized distinct operation is given in Algorithm 40 where the elements of a partition are assigned to a set in order to eliminate duplicates. This algorithm is applied on each partition of the data (axioms) which are in the form of key-value pairs.

```plaintext
Algorithm 40: customizedDistinct(List[(Int, Int)] partitions)

Set[(Int, Int)] uniquePartitions;
foreach p in partitions do
    Set s := {};
    foreach e in p do
        s := s ∪ e;
    end
    uniquePartitions := uniquePartitions ∪ s;
end
return uniquePartitions;
```

- **Subtract.** Subtraction between two collections of axioms is required to keep
track of the newly generated axioms in the previous iteration. The default subtract operation of Spark involves moving the data around the nodes. It is sufficient to perform a local subtract since the data is already partitioned on the key. The customized subtract pseudocode is given in Algorithm 41. Subtract operation is replaced by a left-outer join and a filtering operation. For the given two lists of partitions, the first step is to group by key all the values in the second partition. In the next step, we perform a left-outer join to group key-value pairs having the same keys. Now, subtraction becomes a local operation and we can simply do a filter to get the required result. Remember that the filter operation retains the element that returns true for the condition tested.

\[
\textbf{Algorithm 41:} \text{customizedSubtract}(\text{List}[\text{(Int, Int)}] \text{partitions1}, \text{List}[\text{(Int, Int)}] \text{partitions2})
\]

\[
\text{foreach } p \text{ in partitions2 do}
\]

\[
\text{Map}[\text{(Int, Set[Int])}] \text{groupByKeyMap};
\]

\[
\text{foreach } e \text{ in p do}
\]

\[
\text{/* put() does not overwrite the old value, instead it adds to the set of values at the key */}
\]

\[
\text{groupByKeyMap.put(e.key, e.value)};
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{leftJoin := partitions1.leftOuterJoin(groupByKeyMap.asList());}
\]

\[
\text{diffPartitionsCompoundKey := leftJoin.filter((x, (y, z)) \Rightarrow \{ \text{if } (z \neq \text{null}) \neg z.\text{contains}(y) \text{ else true} \});}
\]

\[
\text{diffPartitions := diffPartitionsCompoundKey.map((x, (y, z)) \Rightarrow (x, y));}
\]

\[
\text{return diffPartitions;}
\]

5. **Kryo Serialization.** The process of converting the state of an object into a byte stream is referred to as serialization. This is required to save the object to a persistent storage and to send it across a communication network. Serialization has an impact on the performance of a distributed application. If the serialization procedure is slow or if the number of bytes that a serializer generates is large, it slows down the application. Java serialization, which is the default serialization mechanism, is slow.
Kryo serialization\(^3\) is the other alternative provided by Apache Spark. It is fast and efficient, but has the requirement that all the custom classes in the application have to be registered.

6. **Tuning Parallelism.** Spark parallelism can be adjusted by configuring the number of partitions in a RDD. The number of tasks is same as the number of partitions and each task runs on a core. So if there are less number of partitions, some cores would be idle and in case of more number of partitions, there will too much overhead in managing all the small tasks. Generally, the number of partitions is set to twice or three times the total number of cores in the cluster.

### 6.5 Evaluation

#### 6.5.1 Evaluation Goals

A distributed reasoner should be able to handle large ontologies and offer good speedup with increase in the number of nodes. Experiments were conducted on SparkEL to answer the following questions.

1. Can SparkEL complete the reasoning task on very large ontologies, i.e., the ontologies that cannot be handled by existing single machine reasoners given reasonable amount of heap size?

2. Does SparkEL show reasonable speedup with increase in the number of nodes in the cluster?

\(^3\)https://github.com/EsotericSoftware/kryo
6.5.2 Datasets

We used biomedical ontologies listed in Table 5.2 as datasets for our experiments. The size of these ontologies are also provided. After reasoning, size of the ontologies increase by 3–15 times.

Endocarditis, Not-Galen, GO (Gene Ontology) and NCI (National Cancer Institute ontology) can be obtained from https://wwwtcs.inf.tu-dresden.de/~meng/toyont.html. SNOMED is available from http://www.ihtsdo.org. We made 5x, 11x and 15x copies of SNOMED in order to test the scalability of our system.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>Number of axioms Before Reasoning</th>
<th>Number of axioms After Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endocarditis</td>
<td>20</td>
<td>63</td>
</tr>
<tr>
<td>Not-Galen</td>
<td>8,188</td>
<td>128,160</td>
</tr>
<tr>
<td>GO</td>
<td>28,897</td>
<td>239,659</td>
</tr>
<tr>
<td>NCI</td>
<td>46,870</td>
<td>346,955</td>
</tr>
<tr>
<td>SNOMEDx1</td>
<td>1,138,819</td>
<td>16,866,392</td>
</tr>
<tr>
<td>SNOMEDx5</td>
<td>5,694,095</td>
<td>84,331,960</td>
</tr>
<tr>
<td>SNOMEDx11</td>
<td>12,527,009</td>
<td>185,530,312</td>
</tr>
<tr>
<td>SNOMEDx15</td>
<td>17,082,285</td>
<td>252,995,880</td>
</tr>
</tbody>
</table>

Table 6.3: Number of axioms in ontologies, before and after reasoning.

6.5.3 Results

Table 6.4 contains the reasoning time (in seconds) for some of the popular single machine reasoners such as ELK 0.4.3, jCEL 0.23.2, Snorocket 2.7.6, JFact 4.0.0 and Konclude 0.6.2 on the datasets shown in Table 5.2. ELK and Konclude support multi-threading. In our experiments, we set the threads to the number of cores, which is 4. Heap size of 12GB was given to all the reasoners including the distributed reasoners. Reasoning on each ontology was performed thrice and the average runtime is calculated. Ontology loading time is
Table 6.4: Reasoning times in seconds of various popular single machine reasoners. OOM is out of memory error. ERR stands for parse error.

excluded wherever applicable.

On larger ontologies such as SNOMEDx5, SNOMEDx11 and SNOMEDx15 all the reasoners (except ELK on SNOMEDx5) run out of memory due to the growth in size of ontologies during the reasoning process. In order to handle large ontologies, a scalable approach to reasoning is required.

In Table 6.5 we compare the scalability and performance of two distributed reasoners, DistEL [75] and SparkEL. DistEL is our previous work on scalable reasoning and currently the only available distributed $\mathcal{EL}^+$ reasoner. Starting with 4 nodes, both these reasoners are run on increasing number of nodes, going up to 64 nodes. Note that, in all these cases, there is an additional master node that schedules jobs on the worker (slave) nodes. Table 6.5 consists of small (upper half of the table), medium and large (lower half of the table) ontologies. Since it is not useful to test the scalability of the distributed reasoners on small ontologies (Endocarditis, Not-Galen, GO and NCI), they are run only on 4 nodes. Other ontologies are run on clusters of varying size, starting with 4 nodes and going up to 64 nodes. Figure 6.1 shows the runtime of DistEL and SparkEL on smaller ontologies such as Endocarditis, Not-Galen, GO and NCI. Except Endocarditis, which is a very small ontology, on all the other ontologies, SparkEL is faster than DistEL. A distributed framework such as Apache Spark and Hadoop take some time for the initialization of processes across the cluster. So on small datasets such as Endocarditis, it would actually take more time
<table>
<thead>
<tr>
<th>Ontology</th>
<th>Distributed Reasoner</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endocarditis</td>
<td>DistEL</td>
<td>0.935</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SparkEL</td>
<td>18.708</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not-Galen</td>
<td>DistEL</td>
<td>89.245</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SparkEL</td>
<td>79.112</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GO</td>
<td>DistEL</td>
<td>76.970</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SparkEL</td>
<td>47.870</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NCI</td>
<td>DistEL</td>
<td>136.244</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SparkEL</td>
<td>66.715</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNOMEDx1</td>
<td>DistEL</td>
<td>853.818</td>
<td>884.492</td>
<td>719.573</td>
<td>681.679</td>
<td>682.212</td>
</tr>
<tr>
<td></td>
<td>SparkEL</td>
<td>579.181</td>
<td>492.907</td>
<td>576.748</td>
<td>660.894</td>
<td>OOMb</td>
</tr>
<tr>
<td>SNOMEDx5</td>
<td>DistEL</td>
<td>4327.974</td>
<td>4310.157</td>
<td>3611.559</td>
<td>3369.016</td>
<td>3353.389</td>
</tr>
<tr>
<td></td>
<td>SparkEL</td>
<td>OOMa</td>
<td>OOMa</td>
<td>OOMa</td>
<td>1027.745</td>
<td>OOMb</td>
</tr>
<tr>
<td>SNOMEDx11</td>
<td>DistEL</td>
<td>OOMa</td>
<td>OOMa</td>
<td>7801.485</td>
<td>7321.253</td>
<td>7323.357</td>
</tr>
<tr>
<td></td>
<td>SparkEL</td>
<td>OOMa</td>
<td>OOMa</td>
<td>OOMa</td>
<td>OOMa</td>
<td>OOMa</td>
</tr>
<tr>
<td>SNOMEDx15</td>
<td>DistEL</td>
<td>OOMa</td>
<td>OOMa</td>
<td>10695.888</td>
<td>10007.392</td>
<td>9907.691</td>
</tr>
<tr>
<td></td>
<td>SparkEL</td>
<td>OOMa</td>
<td>OOMa</td>
<td>OOMa</td>
<td>OOMa</td>
<td>OOMa</td>
</tr>
</tbody>
</table>

Table 6.5: Reasoning times in seconds of distributed reasoners, DistEL and SparkEL. OOMa represents out of memory on the worker (slave) nodes and OOMb represents out of memory on the driver (master) node.

...for initialization than reasoning over the ontology. Another interesting thing to note here is that the reasoning time on GO is lesser than the one on Not-Galen even though the latter is smaller in size compared to the former. This emphasizes the point discussed in Section 2.5.1 that size is not the only factor that determines the reasoning runtime. In this case, majority of GO axioms are simple subclass axioms whereas Not-Galen has several role chain axioms. The rule working on role chain axioms takes longer than the one applied on simple subclass axioms.

### 6.5.4 Discussion

Comparing the distributed reasoners with single machine reasoners such as ELK would be unfair. While the single machine reasoners are definitely faster, the distributed reasoners...
Figure 6.1: Reasoner runtime in seconds for DistEL and SparkEL on various ontologies (smaller is better). These are run on a 4 node cluster

(even DistEL) are more scalable. So we limit our discussion to comparison between DistEL and SparkEL.

In all the cases where SparkEL completes the reasoning task, it is faster than DistEL. This difference in runtime is more prominent in large ontologies such as SNOMED CT (see Figure 6.2) and its multiples. In some cases, such as SNOMEDx5 on 32 nodes, it is 3 times faster. On the other hand, DistEL completes the reasoning task given sufficient number of nodes. There is clearly a trade-off between memory and runtime here, where SparkEL consumes more memory and is faster but does not handle larger ontologies. DistEL on the other hand, uses lesser memory per node but is slower. In SparkEL, out of memory errors occur both on the driver as well as the worker nodes (OOMb and OOMa from Table 6.5). Driver node keeps track of all the task scheduling information. As the number of nodes increase, the number of partitions and hence the number of tasks also increase. Since the reasoning algorithm is iterative, several tasks are generated. This eventually overwhelms the driver and results in out of memory error. Following are the reasons for the worker side OOM error:

- Apache Spark does not use the entire given heap space for computation. The heap space is divided among spark memory, user memory and reserved memory (300MB).
This division of memory is shown in Figure 6.3. Spark memory is in turn divided into storage memory and execution memory. Storage memory is used to store cached data and execution memory is used for operations such as join, sort, aggregation etc. User data structures are stored in user memory. In our experiments, for the 12GB of heap space given to each worker node, only $\approx 4.38$GB is available to execution memory. For operations on large datasets, this is not sufficient.

- According to Apache Spark’s computing model, in an iterative algorithm, if data from previous iteration is not cached, then it is recomputed in the current iteration. This happens if any computation in the current iteration is dependent on previous iteration’s data. For our reasoning algorithm, this lead to the caching of multiple copies of subsets of axioms in each iteration. Although these copies are removed from cache at the end of each iteration, storage memory usage spikes up during an iteration. For larger datasets, this spike is higher and would affect the space available for execution memory.

- Data skew results in improper load balance across the cluster. In the case of SNOMED (and its copies), $R(r) = \{(X_1, Y_1), (X_2, Y_2), \ldots\}$ sets are skewed especially if $r$ is...
Figure 6.3: Memory management in Apache Spark. Image obtained from https://0x0fff.com/spark-memory-management/

taken as the key. Data distribution across the cluster is based on the key. Since joins on $R(r)$ sets happens on key $r$, there is no alternative. Data skew leads to increased memory pressure on some nodes.
7 Comparison of Approaches

In this chapter, we compare and contrast all the four distributed $\mathcal{EL}^+$ reasoning systems discussed in the dissertation – MR-EL (Chapter 3), DQuEL (Chapter 4), DistEL (Chapter 5) and SparkEL (Chapter 6). We also offer recommendations on the best reasoner to choose in different scenarios.

7.1 Flynn’s Taxonomy Revisited

Flynn proposed a taxonomy for computer architectures [29, 26]. The four classifications that he proposed are based on the number of instruction streams and data streams available in the architecture. These two types of streams can be classified based on their combinations.

1. **SISD**. It stands for Single Instruction stream, Single Data stream. This architecture does not support any parallelism. Each instruction is processed sequentially on a single unit of data.

2. **SIMD**. It stands for Single Instruction stream, Multiple Data stream. In this case, a single instruction is applied on multiple data streams. GPU is an example of such architecture.

3. **MISD**. It stands for Multiple Instruction stream, Single Data stream. Multiple instructions are applied on a single stream of data. There are not many machines that
come under this category.

4. **MIMD.** It stands for Multiple Instruction stream, Multiple Data stream. In this case, multiple instructions can be applied on multiple data streams by independent processors. Large scale parallel machines come under this category.

Analogous to this taxonomy, by replacing the term *Instruction* with *Rule*, the four approaches to distributed $\mathcal{EL}^+$ reasoning can be classified into two.

1. **SRMD.** In this architecture, at any point of time in the system, all the nodes in the cluster would be working on a single rule but this particular rule is applied on multiple pieces of the ontology (axioms). This is depicted in Figure 7.1.

![Figure 7.1: Single Rule, Multiple Data architecture](image)

Among the four approaches, MR-EL and SparkEL systems come under this classification. Rules are applied sequentially one after the other. Both these systems make use of existing distributed computing frameworks (Hadoop’s MapReduce and Apache Spark).
2. **MRMD.** In this architecture, all the nodes in the cluster would be working on all the rules and these rules are applied on multiple pieces of the ontology. This is depicted in Figure 7.2.

![Figure 7.2: Multiple Rule, Multiple Data architecture](image)

DQuEL and DistEL systems come under this classification. In this case, each node acts as a complete reasoner (since it can process all the rules) but the logical consequences generated by a single machine are incomplete when taken in isolation.

### 7.2 Characteristics of the four distributed reasoners

The four distributed reasoners, MR-EL, DQuEL, DistEL and SparkEL are compared with respect to some important characteristics. This comparison is summarized in Table 7.1. Following characteristics are used in the comparison.

i) **Axiom representation.** Axioms are generally serialized in the form of XML. But this is not suitable for efficient storage and retrieval in distributed applications. We use key-value representation for axioms in all the four distributed reasoners. Since it...
is possible to have compound keys and values, the key-value format is sufficient to represent all the $\mathcal{EL}^+$ axioms. The key-value representation is simple to handle and it is also easy to distribute the axioms across the cluster.

ii) **Reasoning approach.** Reasoning approach is determined by how the reasoning rules are applied on the axioms. In MR-EL, DistEL and SparkEL, the rules are applied on the axioms iteratively until no new consequence can be derived. This is known as fixpoint iteration. On the other hand, in DQuEL, a queue is assigned to each concept and depending on the queue contents, appropriate rules are triggered.

iii) **Termination condition.** The terminating condition for the reasoning algorithm depends on the reasoning approach and the distributed framework. In MR-EL, after the end of each iteration where all the rules are applied on the axioms once, another map reduce job is run to remove the duplicates. After this step, if the total number of axioms are same as in the previous iteration, the reasoning process is terminated. The same procedure is followed by SparkEL for termination detection. In DistEL, since Redis supports sets as a built-in data structure, there is no need for a separate duplicate removal step. At the end of each iteration, every rule process sends out a message indicating whether any new axiom has been added or not. If none of them added any new axiom, reasoning process is terminated. In DQuEL, termination condition is that all the queues across the cluster should be empty.

iv) **Fault-tolerance.** The property of the system to continue its operations even if there are failures such as process getting killed or a node going down is referred to as fault tolerance. In MR-EL and SparkEL, the underlying frameworks, Hadoop and Spark are fault-tolerant. They reassign the failed tasks to other live nodes. On the other hand, DQuEL and DistEL are not fault-tolerant. If a process dies unexpectedly, the whole system halts.

v) **Axiom distribution.** The partitioning scheme used to distribute the axioms of an
ontology across the cluster plays a crucial role in determining the amount of inter-node communication. Ideally, there should be no inter-node communication but for $\mathcal{EL}^+$ rules, this is not possible due to inter-dependencies among the rules. So the goal of a partitioning scheme is to reduce the inter-node communication as much as possible. In MR-EL and SparkEL, HDFS is used as for storage. The location of the content is determined by the HDFS. Application code is sent to the same data location in order to reduce inter-node communication. In DQuEL, since axioms are in key-value form, hash of the key is used to determine the location of the axiom in the cluster. In DistEL, ontology is partitioned based on the axiom type. For example, all axioms of the form $A \sqsubseteq B$ are assigned to a particular node.

vi) **Rule distribution.** If a snapshot of all the nodes in the cluster is taken at a particular moment in time, *rule distribution* indicates the number of rules are applied across the cluster. In the case of the four distributed reasoners, it is either one rule or all rules. As discussed in Section 7.1, MR-EL and SparkEL come under the SRMD category where a single rule is applied on multiple pieces of data across the cluster. DQuEL and DistEL are in the MRMD category where all the rules are applied simultaneously on various pieces of the ontology across the cluster. Based on the experiments conducted on the four distributed reasoners, it is not entirely clear as to which style of rule application is better. In SRMD, since all the nodes are working on one rule, each rule application is completed relatively quickly and the next rule is taken up. On the other hand, in the case of MRMD, there is more work to be done by each node. So the resources in the cluster are utilized more effectively.

vii) **Dynamic load balancing.** Considering that the hardware and the number of resources (cores, RAM etc) on each node in the cluster is the same, the amount of time taken for each rule application is different. Apart from this, the number of axioms that make up each type ($A \sqsubseteq B$, $A \sqsubseteq \exists r.B$ etc) are different. So it is hard to balance
the load equally across the cluster before the start of the reasoning process. If there is no mechanism to balance the load during runtime, then most likely there would be some idle nodes in the cluster. Among the four distributed reasoners, only DistEL supports dynamic load balancing through work stealing. The idle nodes get some work (axioms) from the busy nodes and apply the appropriate rule on the obtained set of axioms.

viii) **Delta computation.** When applying the reasoning rules iteratively, there is no need to apply the rules on the entire dataset in every iteration. This leads to the generation of duplicate axioms. In order to avoid this, it is sufficient to apply the rules on only the newly generated axioms, i.e., axioms that have not been considered for rule application so far. We refer to this as *delta computation*. In MapReduce approach, it is hard to detect the newly generated axioms. In DistEL, a timestamp based approach is followed to keep track of axioms that have not yet been considered. In SparkEL, we subtract the output of each iteration from the output of all the previous iterations to get the axioms that need to be considered in the next iteration. In DQuEL, an axiom is inserted into the queue only if it doesn’t already exist.

ix) **Data replication.** In the case of the four distributed reasoners discussed in this dissertation, data (axiom) replication is used either to reduce the inter-node communication or to support fault-tolerance. All the four distributed reasoners, selectively replicate some of the axioms or parts of the axioms. In the case of MR-EL, DQuEL and DistEL, subrole and role chain axioms are replicated. In SparkEL, parts of axioms of the form \( \exists r. A \sqsubseteq B \) are broadcast. In DistEL, some of \( R \) sets (see Section 2.1.2 for details on \( R \) sets) are selectively replicated on a subset of nodes. HDFS by default replicates data thrice across the cluster. Since MR-EL and SparkEL use HDFS for storage, all the axioms are replicated to achieve fault-tolerance.

x) **Multi-threading.** Having several cores on machine has now become the norm. So
it is important for the applications to make use of all the available cores in order to use the machine effectively. But in some applications, due to data dependencies it is not always possible to use multi-threading. MR-EL and SparkEL can make use of multiple cores because the underlying frameworks, Hadoop and Spark, can run multiple tasks on a machine. The number of tasks (mappers and reducers in the case of Hadoop) is configurable and when set to a multiple of the number of cores, all the cores on a node would be busy running the tasks. Multi-threading has not been implemented in DQuEL and DistEL. This can be considered as part of future work.

xi) **Data compression.** If data size can be reduced then it not only makes it easier to transmit data over the network but also allows us to store more data in the primary and secondary memory. In distributed applications handling large data, compression plays a crucial role when moving data from one node to the other. MapReduce and Spark support various data compression techniques such as Snappy, LZ(O/4/F) and GZip. Although Redis is optimized to use less space, it does not compress the data.

xii) **Programming language.** The speed and development productivity of the application depends on the programming language. Java is a popular language and there are several libraries and frameworks written in it. OWL API, which is written in Java, is used for reading and writing ontologies. We chose Java since it supports many libraries that are required in building a distributed reasoner. It is also easier to build applications in a higher level programming language such as Java and the performance it offers is sufficient for our purpose. MR-EL, DQuEL and DistEL are written in Java. For SparkEL, we chose Scala since Apache Spark is written in Scala and it runs on JVM, which makes it compatible with any of the Java libraries.
### 7.3 Recommendations

Four different distributed $\mathcal{EL}^+$ reasoners were discussed and there are also several single machine reasoners that are suitable for reasoning over $\mathcal{EL}^+$ ontologies. This naturally leads to the question of which reasoner to choose in different scenarios. We consider four characteristics of the environment on which the choice of the reasoner depends. We use the terms small, medium and large for each of four characteristics.

i) **Ontology size.** The number of axioms in an ontology varies from tens to billions. For our purpose, we consider tens to few hundred axioms as a small ontology, few thousands to few millions as a medium sized ontology and anything greater than few million axioms is considered a large ontology. For small and most medium sized
ontologies, a single machine reasoner is sufficient. A distributed reasoner is required in the case of large ontologies.

ii) **Machine type.** Another important factor in choosing a reasoner is the available hardware resources. We consider three different types of machines. The machine on which the reasoner runs could be a laptop or a shared-memory multi-core system or a networked cluster of machines. The machine type dictates whether a distributed reasoner or a shared-memory reasoner can be run on it.

iii) **Machine size.** The third factor that we consider in making the choice of a reasoner is the amount of resources available to a machine. The resource that is of interest to us is the amount of RAM available on the machine and in particular to the reasoner. Typically, this starts at 4GB and on large servers, it can be 64GB or more. We consider a machine with 4GB RAM as a small machine, the one with 16GB RAM is considered a medium sized machine and the machine with RAM 24GB or more is considered a large machine.

iv) **Cluster size.** The number of nodes in the cluster is referred to as cluster size here. The performance of a distributed reasoner depends on the cluster size. We consider a cluster size of 10 nodes to be small and a cluster size 50 nodes is medium sized. If there are more than 100 nodes in a cluster, it is considered large.

In our experiments, we found ELK 0.4.3 to be the best single machine shared-memory reasoner that is not only fast but also scales reasonably well. Among the four distributed reasoners presented in this dissertation, DistEL is the most scalable, and SparkEL is the fastest. We do not consider the other two distributed reasoners because they are either too slow (MR-EL) or do not scale (DQuEL). In Tables 7.2 and 7.3, we make our recommendations for the best reasoner in different scenarios. We categorize our recommendations into the following three based on the machine type.
i) **Laptop.** This is perhaps the most common machine type on which a reasoner is used. We can consider a standard commodity desktop in this category. From our evaluation, ELK is the best reasoner for single machines. If only a laptop is available, we recommend using ELK for reasoning over $\mathcal{EL}^+$ ontologies.

ii) **Shared-memory server.** Although there are more resources (RAM and cores) available on a shared-memory server, it is better to use a parallel reasoner such as ELK rather than a distributed reasoner since a distributed reasoner operates as if its a shared-nothing cluster rather than a large single machine with many cores. On the other hand, reasoners such as ELK can take advantage of the resources available when reasoning over very large ontologies.

iii) **Shared-nothing cluster.** Apache Spark and as a consequence the distributed reasoner SparkEL needs good amount of RAM to complete the reasoning over large ontologies efficiently. So it is not recommended to use SparkEL on small machine types with limited RAM. From Table 6.5, it can be seen that for small ontologies such as Endocarditis, it is better to use DistEL since it has lower startup time compared to SparkEL. For medium sized ontologies such as medium sized ontologies such as Not-Galen, GO, NCI and SNOMED CT, SparkEL has lower runtime. Based on these experiments, we recommend (Table 7.3) that for medium sized ontologies on small, medium or large clusters, it is better to use SparkEL. Large machine types making up a large cluster is the ideal scenario for the use of SparkEL. It can handle large ontologies efficiently.

In order to cover all the possible scenarios in different environments for reasoning over $\mathcal{EL}^+$ ontologies, from the recommendations in the table, it is clear that a distributed reasoner complements a single machine shared-memory reasoner and does not compete with it. So depending on the available machine type, size and cluster size either ELK,
### Table 7.2: Recommendations for a reasoner on single machine based on the ontology size and the machine type available for reasoning

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>Machine Size</th>
<th>Ontology Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>Small</td>
<td>ELK</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>ELK</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>ELK</td>
</tr>
<tr>
<td>Shared-memory server</td>
<td>Small</td>
<td>ELK</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>ELK</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>ELK</td>
</tr>
</tbody>
</table>

### Table 7.3: Recommendations for a reasoner on a shared-nothing cluster based on the ontology size and the machine type available for reasoning

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>Cluster Size</th>
<th>Machine Size</th>
<th>Ontology Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small</td>
<td>Small</td>
<td>DistEL</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>Small</td>
<td>DistEL</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>Small</td>
<td>DistEL</td>
</tr>
<tr>
<td>Shared-nothing cluster</td>
<td>Medium</td>
<td>Small</td>
<td>DistEL</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>Small</td>
<td>DistEL</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>Small</td>
<td>DistEL</td>
</tr>
</tbody>
</table>

DistEL or SparkEL can be used for reasoning over $\mathcal{EL}^+$ ontologies.

### 7.4 The Way Forward

From the comparison of DistEL and SparkEL on various ontologies (Section 6.5.3), it is clear that the performance of DistEL can be improved further and the memory consumption of SparkEL can be improved as well. A customized distributed system for $\mathcal{EL}^+$ reasoning allows us greater flexibility in memory management and performance tuning. Reasoning algorithms of SparkEL are simpler and cleaner compared to DistEL’s algorithms. Using SparkEL’s algorithms along with its multi-threaded functionality and avoiding Spark’s
computing model could lead to a distributed reasoner that not only scales but performs on par with SparkEL. At the same time, a lower level framework that takes care of issues such as communication, synchronization, fault-tolerance etc is required. An idea for such a system is described in Section 10.2.
8 Related Work

In this chapter, we review some of the existing work on scalable reasoning over the Semantic Web languages. Efficient reading and writing of the data are important operations in a distributed computing setup. So selection of an appropriate database is vital as it could lead to performance bottlenecks. In this chapter, we also review some of the currently available distributed storage solutions.

8.1 Scalable Reasoning

There are several language profiles in the Semantic Web that support rule-based reasoning. Other reasoning approaches such as tableau algorithms are not considered here. Scalable reasoning approaches such as parallel shared memory reasoning and distributed shared-nothing reasoning exist. Although our focus is on distributed $\mathcal{EL}^{++}$ reasoning, similar approaches were tried for other Semantic Web languages and description logics. The different rulesets of these language profiles brings in varied challenges.

8.1.1 RDFS

RDFS has a number of inference (entailment) rules including simple, extensional and datatype entailment rules [39]. Almost all of the existing work on scalable RDFS entailment (closure) computation considers only a subset of these rules. RDFS is the profile
on which perhaps the most number of scalable approaches were tried. MARVIN (MAssive RDF Versatile Inference Network) computes the closure of RDF triples using a peer-to-peer model [79]. It uses the divide-conquer-swap strategy where the data is partitioned across the cluster, closure is computed and triples are exchanged among the neighboring nodes. They implement a SpeedDate algorithm that exchanges triples within a restricted set of neighborhood nodes. The system scales up to 64 nodes handling up to 14.9 million triples.

In [106], triples are distributed across the cluster by making a distinction between the schema and instance triples. Finding closure becomes an embarrassingly parallel computation. Schema triples are less in number compared to instance triples. So all the nodes in the cluster have schema triples present locally whereas, instance triples are partitioned equally and allotted to different nodes. Implementation is in C and MPI. Evaluation was done using the data generated from LUBM on a 128 core machine. There are about 650 million triples at the end of closure.

Cray XMT, a massively parallel shared memory supercomputer was used in [33] for RDFS inference, dictionary encoding and SPARQL query processing. All these computations are performed completely in-memory. RDF triples are stored in global hash table. A technique called Sprinkle SPARQL is used to identify matching triples in a SPARQL query. The system scales up to 512 processors handling 20 billion triples.

Parallel shared memory implementation of RDFS closure is shown in [40]. The experiments are run using DBpedia and YAGO datasets on a multi-core CPU as well as a GPU. Although GPU is good at parallel operations, much time is spent on data that needs to be copied between the main memory and graphic memory.

Stream reasoning over RDF data and C-SPARQL query answering using Yahoo S4 is discussed in [43]. Minimization of triples and duplicate triple elimination are mentioned as challenges. The process of finding the closure is based on eventual completeness i.e., the system gradually gathers logical consequences until no new logical consequences can be
derived. System is evaluated on two metrics which are maximum throughput and number of nodes. Up to 8 nodes, it demonstrates linear scalability and high throughput. A throughput of hundreds of thousands of triples per second is achieved. Beyond 8 nodes, throughput does not increase linearly.

DynamiTE [102] is another parallel stream reasoner that has support for incremental materialization i.e., addition and removal of RDF triples to existing knowledge base of RDF triples. When data is added, it uses the parallel version of well-known semi-naive evaluation of Datalog. In case of data removal, two algorithms have been implemented - one with and another without bookkeeping (Delete and Rederive algorithm). Evaluation was done on a single machine with 8 cores. Up to a billion triples are used in the experiments and the throughput obtained is 227k triples/second.

Backward chaining approach is used in [83] for reasoning over minimal RDFS rules on top of 4Store, a distributed RDF storage engine. Terminological (schema) triples are replicated on all the nodes and instance triples are evenly distributed in a non-overlapping manner across the machines in the cluster. In order to process a query, each node rewrites it into multiple queries in order to use the local terminological knowledge. LUBM was used for experiments on a 5 machine cluster. Up to 138 million triples were used in the evaluation.

One of the most commonly used triple data partitioning strategy is to use the terms (subject/predicate/object) in them. This introduces the problem of improper load balancing, since some of terms could be more frequently occurring than others. In [53], this problem is addressed by creating elastic regions in the cluster. Triples are not fixed to particular nodes and a routing bias is created that allows the triples to propagate to neighboring nodes. Whenever a node receives new triples, reasoning is performed again. Experiments were performed using up to 200 million triples on a 64 node cluster.

Distributed hash tables, which are peer-to-peer systems, are used for RDFS reasoning in [46]. Reasoning using both forward and backward chaining is implemented. RDFS
entailment rules are rewritten as datalog rules so as to utilize the optimization techniques available for datalog. This is first of the works that utilized distributed backward chaining for RDFS reasoning. For the experiments, $10^3$ and $10^4$ triples are used. The time taken to store these triples in forward chaining approach is much longer. 100 queries are used to test the query response times. Query answering using forward chaining is much more efficient than backward chaining.

8.1.2 OWL Horst

OWL Horst (also known as pD*) [96] extends RDFS entailment rules to include reasoning with datatypes from a given datatype map D. Rule partitioning and data partitioning strategies are explored in [91] for computing closure over OWL Horst knowledge base. In the case of rule partitioning, each node works on a subset of rules. This might lead to improper load balancing because this partitioning strategy depends on the number of rules. On the other hand, in the case of data partitioning, each subset of the data is assigned to a node. This leads to better load balancing. But for dynamic load balancing, rule partitioning approach is better suited since there are limited number of rules in comparison to data. Several goals for the partitioning strategy such as balanced partitioning, minimizing communication, efficiency (reduce duplicate generation of triples), speed and scalability are defined. Jena reasoner on a cluster of machines is used to evaluate the two partitioning strategies. For data partitioning, graph and hash based along with domain specific partitioning strategies are compared.

WebPIE [100] is a distributed forward chaining reasoner that uses MapReduce approach (Hadoop framework) and supports OWL Horst semantics. Data is partitioned across the cluster and rules are applied in parallel. This system was able to scale linearly up to 100 billion triples on a 64 node cluster. The datasets used were UniProt, LUBM and FactForge.

QueryPIE [103] is a backward chaining distributed reasoner that supports OWL Horst reasoning over large knowledge bases. It can run on a single multi-core machine or on a
cluster of machines. Terminological triples are materialized and are accessible to all the machines in the cluster. A parallel version of QSQ algorithm is used in order to improve the performance of top-down evaluation of rules. Several rules are evaluated in parallel. LUBM, LinkedLifeData and FactForge datasets are used for experiments. Querying over 1 billion triples on a 8 machine cluster can be done in the order of few milliseconds.

Cichlid [35] is a distributed reasoning engine for RDFS and OWL Horst rulesets. It is built on top of the Apache Spark framework. Rules are implemented using the Spark RDD (resilient distributed dataset) programming model. RDD is a fault-tolerant collection of items that can be operated in parallel. Join processing, transitive closure computation and equivalent relation computation are optimized among other things. A 17 node cluster is used to perform distributed reasoning over DBpedia, LUBM and WordNet datasets. Compared to WebPIE, this implementation is 10 times faster.

8.1.3 OWL 2 RL

An OWL 2 RL inference engine is built in Oracle database system [51]. A hybrid (memory and disk based) approach was developed for building owl:sameAs cliques and other equivalence relations such as owl:equivalentClass and owl:equivalentProperty. Multi-pattern rules are run in parallel. Other optimization techniques such as lazy duplicate elimination and dynamic semi-naive evaluation are used. Experiments with 1 billion triples for incremental reasoning (when 10,000 triples are added) shows that it takes only a few minutes to do the inference update.

SAOR [44] is a distributed reasoner that has support for RDFS, OWL Horst and OWL 2 RL. Partial-indexing approach is reformulated and generalized to arbitrary rulesets. Terminological data is separate from rest of the triples. Generic rule optimizations based on template rules are also used. Rule index is developed in order to efficiently identify which rules to apply for a particular triple. Rule dependency graph is used to check the order of application of rules. This approach is evaluated on a 8 machine cluster and 1.12 billion
triples. Reasoning time is 3.35 hours and the output consists of 1.58 billion triples.

### 8.1.4 OWL 2 EL

There is very little work implemented, evaluated and published on distributed approaches to OWL 2 EL reasoning. Ours is the first such work that discusses rule-based distributed $\mathcal{EL}^{++}$ (description logic underpinning for OWL 2 EL) reasoning.

A distributed resolution technique for $\mathcal{EL}^+$ classification is presented in [84]. MapReduce framework is used to check satisfiability of expressive ontologies. Distributed resolution is used to check satisfiability for a given set of axioms. Optimization for avoiding repetitive rule application and load balancing strategy has been suggested. However, no implementation or evaluation of this approach is discussed.

Though not distributed, parallelization of OWL 2 EL classification has been studied in [47, 81]. Optimized consequence-based procedure is used for classification. Axioms are assigned to contexts which can be processed independently using multiple threads. In the evaluation, the largest ontology used is SNOMED CT and it can be classified in 5 seconds. Classifying EL ontologies on a single machine using a database has been tried in [24].

Distributed Fuzzy-$\mathcal{EL}^+$ using MapReduce approach has been tried in [110, 111]. No evaluation has been provided. This is an extension to the MapReduce approach for $\mathcal{EL}^+$ ontology classification from [76].

### 8.1.5 General Rule-Based Reasoning

There are some existing generic rule-based scalable reasoning approaches. RETE implementation on GPUs for RDFS and OWL Horst rulesets is shown in [80]. Memory efficiency of the RETE implementation has been improved and a compressed triple-index structure has been introduced which is used to identify duplicate triples. Also, triples are encoded using differential encoding and variable byte coding. Evaluation has been done on a laptop
using 1 billion triples.

Another alternative is to convert different rulesets into datalog rules. A parallel implementation of datalog programs with application to RDFS and OWL 2 RL rules is shown in [72]. Initially, all the facts are ordered according to a strict total order. A global iterator is provided on top of this ordered table. A free thread can pick a fact and match it to the atoms in all rules. But while picking the facts, their total order has to be respected. This process is repeated until no new facts are generated by the threads. The global order helps in avoiding repeated computations. Implementation of this system is called RDFox. Experiments were run on a machine with 128GB RAM, 16 physical and 32 virtual cores. In the evaluation, OWL 2 RL rules have been translated to datalog program. LUBM dataset is used for experiments. With 16 threads, RDFox is 13.9 times faster than with just 1 thread. The performance improvement through parallelization decreases with more number of threads after this point. However, with LUBM5K (around 691 million triples), it results in memory exhaustion.

8.2 Distributed Databases

A distributed file system [98, 30, 14, 107] provides a coherent and shared access to the same set of files to a cluster of machines. It provides a scalable and highly available file system. They aim to be transparent, i.e., access to a distributed file system should be invisible and be same as accessing a local file system. Transparency is provided at multiple levels, i.e., location, access, replication, migration, failure.

A NoSQL store (database) [82, 37, 18] is generally interpreted as an abbreviation for *Not only SQL*. It differs from traditional relational databases in terms of support for schema, ACID properties and querying. Generally, NoSQL stores support a very flexible schema and minimal query operators. One of key features of a NoSQL store is the ease with which data can be scaled horizontally [68], i.e., data is partitioned across a cluster of machines.
Several NoSQL stores are available and they differ in the data model and in the implementation of CAP theorem [15, 94]. The CAP theorem states that from among the following three desirable features, a distributed database system can implement at most two. 1) **Consistency** of data in a distributed database is equivalent to having a single up-to-date copy of the data, i.e., different nodes in the cluster should not give different copies of the same data. 2) **Availability** means that when a non-failing node in a cluster receives a request, a response is returned within a reasonable amount of time and 3) **Partition tolerance** implies that even if there is a communication breakdown among the nodes in the cluster, thus splitting the nodes into groups, the system continues to function.

There are around 150 NoSQL databases (http://nosql-database.org) that are available currently. We will briefly describe some of the popular ones on the basis of the supported data model.

1. **Key-Value Stores**

   In a key-value store, a unique key is mapped to a set of values, i.e., a key is used to identify a set of values. Key lookups are very fast but the schema of the data is restricted to the simple key-value pairs. Querying is generally limited to simple operations such as `put` and `get`. The main advantage of a key-value store is its simplicity, scalability and fast look-ups.

2. **Column-Oriented Databases**

   A column-oriented database [1] stores data in terms of columns rather than rows as is the case with traditional relational database systems. They vertically partition the database into a collection of columns. Each column is stored separately on disk. When querying the database, instead of reading the entire rows, only the required attributes can be read. Column stores compute aggregates efficiently since aggregation is generally performed over similar data items. Data warehousing is also efficient with column stores.
3. **Document Databases**

A document database stores and manages a group of documents. Each document is semi-structured and can contain sets of key-value pairs similar to a JSON object. Schema of the document is flexible, i.e., no two documents need to have the same structure. Compared to key-value stores, supported querying operations are fairly extensive. Any field of the document can be indexed and a set of documents can be easily partitioned across a cluster of machines based on the keys in the document.

4. **Graph Databases**

A graph database [3] stores and queries graphs along with its vertices, edges and properties. Data is represented in terms of graphs and it is queried using graph manipulation operations such as finding paths, neighborhoods, subgraphs, graph partitioning etc. Graph allows a richer representation of data compared to other data models discussed here. Relationships are the core aspect of this data model and they are generally the first class citizens. However, queries on graphs are generally expensive and graphs can quickly grow to very large size having billions of edges. There is on going research work on distributed graph processing [87, 64, 32, 109].

Some of the popular NoSQL stores along with its data model and CAP theorem compliance is given in Table 8.1 [37].

### 8.3 Limitation

Based on the literature survey presented here, following is the missing piece.

Scalable reasoning approaches have been proposed for RDFS, OWL 2 RL and OWL Horst. The inference rules for all these language profiles can be parallelized i.e., there are several rules that can be applied independently of each other. But this is not the case for the
<table>
<thead>
<tr>
<th>NoSQL Store</th>
<th>Data Model</th>
<th>CAP Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bigtable [?]</td>
<td>Column-oriented database</td>
<td>CP</td>
</tr>
<tr>
<td>Hypertable (<a href="http://hypertable.org">http://hypertable.org</a>)</td>
<td>Column-oriented database</td>
<td>CP</td>
</tr>
<tr>
<td>HBase (<a href="http://hbase.apache.org">http://hbase.apache.org</a>)</td>
<td>Column-oriented database</td>
<td>CP</td>
</tr>
<tr>
<td>MongoDB (<a href="https://www.mongodb.org">https://www.mongodb.org</a>)</td>
<td>Column-oriented database</td>
<td>CP</td>
</tr>
<tr>
<td>Redis (<a href="http://redis.io">http://redis.io</a>)</td>
<td>Key-Value store</td>
<td>CP</td>
</tr>
<tr>
<td>MemcachedDB (<a href="http://memcachedb.org">http://memcachedb.org</a>)</td>
<td>Key-Value store</td>
<td>CP</td>
</tr>
<tr>
<td>BerkeleyDB (<a href="http://bit.ly/1QtyUKG">http://bit.ly/1QtyUKG</a>)</td>
<td>Key-Value store</td>
<td>CP</td>
</tr>
<tr>
<td>Voldemort [95]</td>
<td>Key-Value store</td>
<td>AP</td>
</tr>
<tr>
<td>Kyoto Cabinet (<a href="http://bit.ly/1K1BxPp">http://bit.ly/1K1BxPp</a>)</td>
<td>Key-Value store</td>
<td>AP</td>
</tr>
<tr>
<td>Riak [50]</td>
<td>Key-Value store</td>
<td>AP</td>
</tr>
<tr>
<td>CouchDB (<a href="http://couchdb.apache.org">http://couchdb.apache.org</a>)</td>
<td>Document database</td>
<td>AP</td>
</tr>
</tbody>
</table>

Table 8.1: Some popular NoSQL stores along with their data model and compliance with CAP theorem

Inference rules of $\mathcal{EL}^+$, which is why it is harder to find scalable approaches for ontologies in $\mathcal{EL}^+$ profile. There are already $\mathcal{EL}^+$ ontologies that are very large and cannot be handled by single machine reasoners.

In this dissertation, we address the missing piece of finding scalable approach for $\mathcal{EL}^+$ reasoning.
9 Conclusion

With the ever increasing data growth rates, in order to make sense of the data, structure and meaning can be added to it. In the context of Semantic Web, OWL and RDF can be used to represent the domain knowledge as well as add structure to the data. In comparison to RDF, OWL can be used to represent more expressive and complex relationships. In this dissertation, we focus on a particular profile of OWL 2 known as OWL 2 EL as well as its underlying description logic $\mathcal{EL}^+$. 

Domain knowledge created using OWL is called an ontology. Reasoning is an operation that can be performed on an ontology. But the existing reasoners cannot handle large ontologies as they are constrained by the resources available on a single machine. In this dissertation, we explore distributed approaches to $\mathcal{EL}^+$ reasoning.

We discussed four distributed $\mathcal{EL}^+$ reasoners, namely MR-EL, DQuEL, DistEL and SparkEL. They vary in terms of ontology and rule partitioning strategies used. We also provide a comparison of all the four reasoners, along with the recommendation of the best choice of reasoner for different scenarios.
10 Future Work

There are several directions in which future work can be pursued. We propose a general distributed rule-based reasoning framework that can take any ruleset as input and apply it on the axioms. It takes care of the distribution of rules and ontology in an efficient manner. We also propose alternate approach for scalable and efficient distributed $\mathcal{EL}^+$ reasoning. In a distributed environment, it is important to utilize resources optimally so as to get the best performance and also to save the costs. But it is hard to find the right cluster size for a given ontology. We explore all these future directions along with extending our methodology to other OWL profiles in this chapter.

10.1 Rule-Based Distributed Reasoning Framework

An ontology can belong to any of the several description logic languages such as $\mathcal{EL}^{++}$, OWL Horst, RDFS etc. Reasoning over ontologies is performed using a set of rules that vary with each profile. Existing solutions are tuned towards a particular ruleset. A unified distributed reasoning framework can be developed that can work on any given ruleset [77]. This framework can, not only handle the aforementioned language profiles but also avoids the need to develop a customized scalable approach for any new ruleset.

Some of the challenges in the design and implementation of a rule-based distributed reasoning framework are discussed here.
1:  s p o (if o is literal)  ⇒  _:n rdf:type rdfs:Literal
2:  p rdfs:domain x & s p o  ⇒  s rdf:type x
3:  p rdfs:range x & s p o  ⇒  o rdf:type x
4a: s p o
4b: s p o
5:  p rdfs:subPropertyOf q & q rdfs:subPropertyOf r  ⇒  p rdfs:subPropertyOf r
6:  p rdf:type rdfs:Property  ⇒  p rdfs:subPropertyOf p
7:  s p o & p rdfs:subPropertyOf q  ⇒  s q o
8:  s rdf:type rdfs:Class  ⇒  s rdfs:subClassOf rdfs:Resource
9:  s rdf:type x & x rdfs:subClassOf y  ⇒  s rdf:type y
10: s rdf:type rdfs:Class  ⇒  s rdfs:subClassOf s
11: x rdfs:subClassOf y & y rdfs:subClassOf z  ⇒  x rdfs:subClassOf z
12: p rdf:type rdfs:ContainerMembershipProperty  ⇒  p rdfs:subPropertyOf rdfs:member
13: o rdf:type rdfs:Datatype  ⇒  o rdfs:subClassOf rdfs:Literal

Table 10.1: RDFS closure rules

10.1.1 Rule dependency analysis

If the input to a rule $R_i$, depends on the output of another rule $R_j$, then the rule $R_i$ is dependent on $R_j$. The more independent the rules are, better can be the rule distribution among the nodes in the cluster.

A rule dependency graph can be constructed in order to determine the inter-dependency among the rules. Each vertex represents a rule and an outgoing edge between vertex $v_i$ and $v_j$ represents the dependency of vertex $v_i$ on $v_j$. Isolated vertices are independent of each other and can be executed in parallel. For each vertex $v_i$, all the vertices that are reachable from it are dependent on each other.

RDFS rules from [101] are shown in Table 10.1 and its dependency graph is shown in Figure 10.1. There are several rules that are independent and can be given to separate nodes in the cluster. Rules 5, 6 and 7 are dependent on each other and can be grouped together i.e., all the three rules can be executed by one node. Same holds for the group of rules 9, 2, 3, 10, 11.

On the other hand, rules for $\mathcal{EL}^{++}$ [6] are highly inter-dependent and are difficult to
parallelize.

10.1.2 Rule implementation

Interpreting different rule sets will be very difficult for the framework. Instead, rule sets should be converted to a common domain specific language (DSL) that is supported by the framework. This DSL should be able to define the vocabulary, syntax and semantics of the language to be used.

For the choice of DSL, there are some options. 1) general purpose rule languages such as RETE, Datalog and Prolog. 2) or a custom DSL for the rules supported by the framework. DSL should support the declaration of variables and constants in the rules.

10.1.3 Possible Evaluation Plan

The rule-based distributed reasoning framework can be evaluated along the lines of adaptation, extension and scalability.

- There are several existing specialized and scalable reasoners for rule sets such as RDFS (WebPIE, Cichlid), OWL Horst (QueryPIE) and OWL 2 EL (DistEL). The framework should be able to handle these rule sets. The performance of the general purpose framework in comparison to the specialized ones remains to be seen.
• The framework should be able to take in a new ruleset [89] and provide sound and complete inference over the given data.

• Scalability of the framework will be evaluated with increasing number of nodes on large ontologies.

10.2 ELder: A Concurrent and Distributed Reasoner

10.2.1 Rule Partitioning Strategy

Each of the six rules from Table 2.2 produce output that is either $S(X)$ or $R(r)$, i.e., each rule contributes to either $S$ or $R$. We refer to all such rules that contribute to $S$ as $S$ group, denoted by $S_g$ and all such rules that contribute to $R$ are called $R$ group, denoted by $R_g$. Note that, in $S_g$, all the rules depend only on $S$, i.e., their input is $S$. An exception to this rule is $R_4$, that depends not only on $S$ but also $R$. In group $R_g$, all the rules depend only on $R$ with the exception of rule $R_3$ that also depends on $S$. These two groups, along with the rules that make up each group is given in Figure 10.2.

The six rules can be partitioned into two groups ($S_g, R_g$) of three rules each. If each group is considered as a subgraph, with each rule making up a vertex and the dependencies among the rules as edges then this partitioning strategy is similar to vertex partitioning with minimum cut on the edges joining the two subgraphs.

10.2.2 Termination

Each group, $S_g$ and $R_g$, handle termination separately. The termination procedure followed by $S_g$ and $R_g$ is given in the flowchart of Figure 10.3, with $S_g$ termination on the left and $R_g$ termination on the right.
Change detection in sets $S$ and $R$

Each group consists of three rules. For the sake of simplicity, let us assume that each rule application is taken care of by one process and another process handles all the $S$ (or $R$) sets, i.e., there are four processes in total in each group. In order to determine the changes made by the three rule processes, they need to come to a consensus. This can be achieved by broadcasting the status (update/no update) message by each rule process to rest of the processes in the group. After receiving the status message, each rule process can wait for other rule processes and continue with the rule application. The other option is to not wait for the remaining processes and continue with the rule application.

- **Barrier synchronization**: The three rule processes wait until all of them complete and then apply the rules again on the axioms.

- **Without barrier**: Lets say R1 finishes before R2 and R3. Then R1 process continues with the next iteration (after broadcasting the status message) without waiting for R2 and R3, i.e., irrespective of the status of R2 and R3, R1 speculates that there might be some changes and goes to the next iteration. If it turns out that there are no changes
Figure 10.3: Flowchart that indicates the termination procedure of $S_g$ and $R_g$ made by R2 or R3 then R1 process can be aborted.

10.2.3 Advantages of this partitioning strategy

In comparison to DistEL, this partitioning and termination strategy has the following advantages.

- Termination: In DistEL, there is a barrier on all the six rule processes. So there is a chance for several rule processes to wait for the slowest one. In ELder, barrier is applied only on three rule processes. An alternative to achieve termination without barrier is also discussed.

- Grouping: It is sufficient for $S_g$ and $R_g$ groups to communicate within the group. Due to the partitioning strategy, it is not required to broadcast messages to all the processes in the cluster.
• Concurrency: All the rule processes work concurrently and exchange messages for communication.

• No database: All the data structures are in-memory. An external database is not used in this work. If processes run out of memory, additional node(s) is added to the cluster in order to share the load.

### 10.2.4 Actor Model

An actor is a concurrent computational entity that encapsulates a state and behavior. The only way to communicate with an actor is by passing a message to it. An actor responds to a message based on the defined behavior. This model provides the basis to develop highly concurrent distributed systems. Akka ([http://akka.io/](http://akka.io/)) is a popular implementation of the actor model. Following features of Akka are helpful in this work.

- **Akka cluster**: Akka cluster is a fault tolerant peer-to-peer network of nodes with no single point of failure. Nodes can join and leave the cluster dynamically.

- **Cluster sharding**: This is useful to distribute the actors across the cluster. Location transparency is provided, i.e., actors can be communicate with each without the need to know their actual physical locations in the cluster.

- **Node roles**: Nodes in the cluster can be assigned roles. For example, in this work, we can assign the roles of $S_g$ and $R_g$ to nodes. Roles can be assigned to actors during their creation and depending on these roles, actors can be deployed on the nodes with corresponding roles.

- **Light weight**: An actor is lighter than a thread. Akka claims that around 2.5 million actors can be accommodated in 1GB of heap space.
10.3 Optimal Resource Provisioning for Ontologies

For a given ontology, what is size of the cluster that provides optimal reasoning time? This depends on a variety of factors such as the size of the ontology, the type of axioms it has, the rule and ontology partitioning strategies. Allocating the maximum number of nodes available to the reasoning task does not necessarily result in optimal reasoning time. Optimizing resource provisioning also helps in minimizing the financial cost in running the cluster. There is work on optimal resource provisioning for other forms of data on specific platforms such as Hadoop’s MapReduce [20]. We believe extending this work to ontologies on any general distributed system would be beneficial as well as challenging.

10.4 Distributed Reasoning over other OWL 2 Profiles

In this dissertation, only OWL 2 EL profile was considered for distributed reasoning. An interesting future direction could be to use some of the techniques and optimizations for distributed $\mathcal{EL}^+$ reasoning in other profiles such as OWL 2 RL, OWL 2 QL and OWL DL. In particular, the approach used for SparkEL could be utilized for reasoning over OWL 2 RL and OWL 2 QL. From [12], it is clear that reasoning over OWL 2 RL and OWL 2 QL is possible by implementing rulesets. Also, SAOR [44] demonstrates that distributed reasoning is possible over OWL 2 RL. Apache Spark, which is the underlying framework in SparkEL, supports operations such as join, and set operations such as union, intersection and subtract. So it would not be hard to implement rules from OWL 2 RL and OWL 2 QL in Apache Spark. Also, several optimizations from SparkEL would also be useful for RL and QL rulesets. Compared to OWL 2 EL, OWL 2 RL and OWL 2 QL rulesets have lesser inter-dependencies, so it should be more amenable to parallelization.

Reasoning on OWL 2 DL ontologies is intractable and has a complexity of N2ExpTime. Considering that the reasoning runtime of distributed reasoners in $\mathcal{EL}^+$ with polynomial
time complexity haven’t been as good as shared-memory single machine reasoners, it is not recommended to investigate distributed reasoning approaches for OWL 2 DL with exponential time complexity. Although there are tableau based algorithms for reasoning over OWL 2 DL, to the best of our knowledge, there are no rule based saturation techniques. So techniques discussed in this dissertation would, most likely not be useful for reasoning on OWL 2 DL ontologies.
A Termination, Soundness and Completeness Proofs

We now prove that the classification of ontologies using the completion rules of Table A.1 terminates and is sound and complete. This follows along the line of the corresponding proofs in [5].

$$\textbf{Axiom Rn} \Rightarrow \textbf{Action}$$

\begin{align*}
A \sqsubseteq B \quad & \Rightarrow \quad U[B] \cup= U[A] \\
A_1 \cap \cdots \cap A_n \sqsubseteq B \quad & \Rightarrow \quad U[B] \cup= U[A_1] \cap \\
& \quad \cdots \cap U[A_n] \\
A \sqsubseteq \exists r.B \quad & \Rightarrow \quad R[r] \cup= \{(X, B) \mid X \in U[A]\} \\
\exists r.A \sqsubseteq B \quad & \Rightarrow \quad \text{generate new } \exists r.Y \sqsubseteq B \\
& \quad \text{for each } Y \in U[A] \\
\exists r.Y \sqsubseteq B \quad & \Rightarrow \quad U[B] \cup= \{X \mid (X, Y) \in R[r]\} \\
X \sqsubseteq \exists r.Y \quad & \Rightarrow \quad U[\bot] \cup= \{X \mid Y \in U[\bot]\} \\
r \sqsubseteq s \quad & \Rightarrow \quad R[s] \cup= R[r] \\
r \circ s \sqsubseteq t \quad & \Rightarrow \quad R[t] \cup= \{(X, Z) \mid (X, Y) \in R[r], (Y, Z) \in R[s]\} \\
\end{align*}

Table A.1: Completion Rules for classification
A.1 Termination

On the axioms of an ontology, the rules of Table 5.1 can only be applied a polynomial number of times, and each rule application is polynomial.

Proof. It is readily checked that the cardinality of $BC_O$ and $N_R$ is linear in the size of the axioms in the ontology. Each rule application performed by the algorithm adds a new element of $BC_O \cup \{ \bot \}$ to a set $U[X]$, for some $X \in BC_O$, or a new tuple $(X, Y) \in BC_O \times BC_O$ to a relation $R[r]$, for some $r \in N_r$. Since no rule removes elements of these sets/relations, the total number of rule applications is polynomial. It is readily checked that each rule application can be performed in polynomial time.

A.2 Soundness

Proposition A.2.1. We will use $s_0 \ldots s_n$ to indicate a sequence of applications of the completion rules. For each $0 \leq i \leq n$, there corresponds a mapping $U_i[A]$ and $R_i[r]$ for each concept name $A$ and role $r$. The start $s_0$ of the sequence corresponds to the mappings initialized as described previously in the paper. $U[A]$ and $R[r]$ (without the subscripts) indicates the mappings after the sequence has reached a fixpoint. We claim the following:

For any concept names $C$ and $D$ and role name $r$, and for any model $I$ of the KB:

1. If $C \in U[D]$, then $C^I \subseteq D^I$.

2. If $(C, D) \in R[r]$, then $C^I \subseteq (\exists r.D)^I$.

Proof. We induct on the sequence $s_0, \ldots$, showing for each $i \geq 0$, if $C \in U_i[D]$, then
\[ C^\mathcal{I} \subseteq D^\mathcal{I}, \text{ and if } (C, D) \in R_i[r], \text{ then } C^\mathcal{I} \subseteq (\exists r.D)^\mathcal{I}. \]

**base case** If \( C \in U_0[D] \) then either \( C = D \) or else \( C = \bot \). Each \( R_0[r] \) is empty. As such, the claim holds for \( i = 0 \).

Suppose the claim holds for all \( i < n \) and that \( C \in U_n[D] \) (and also that \( C \notin U_{n-1}[D] \)). As such, one of the following rules must have been applied: R1, R2, R5, R6.

We consider each rule in turn.

**R1** Suppose \( C \) is added to \( U_n[D] \) via rule R1. Then \( C \) appears in \( U_{n-1}[E] \) for some concept \( E \) and axiom \( E \subseteq D \) appears in the KB. Since \( \mathcal{I} \) is a model of the KB, \( E^\mathcal{I} \subseteq D^\mathcal{I} \). By the inductive hypothesis, \( C^\mathcal{I} \subseteq E^\mathcal{I} \). From this, it follows that \( C^\mathcal{I} \subseteq D^\mathcal{I} \).

**R2** Suppose \( C \) is added to \( U_n[D] \) via rule R2. Then \( C \in U_{n-1}[E_1], \ldots, C \in U_{n-1}[E_m] \) for concepts \( E_1, \ldots, E_m \) and axiom \( E_1 \cap \ldots \cap E_m \subseteq D \) appears in the KB. By the inductive hypothesis, \( C^\mathcal{I} \subseteq E_i^\mathcal{I} \) for each \( 0 \leq i \leq m \). Since \( \mathcal{I} \) is a model of the KB, \( E_1^\mathcal{I} \cap \ldots \cap E_m^\mathcal{I} \subseteq D^\mathcal{I} \). From this, it follows that \( C^\mathcal{I} \subseteq D^\mathcal{I} \).

**R5** Suppose \( C \) is added to \( U_n[D] \) via rule R5. Then \( (C, Y) \in R_{n-1}[r] \) for some \( r \) and \( \exists r.Y \subseteq D \) is an axiom of the KB. By the inductive hypothesis, \( C^\mathcal{I} \subseteq (\exists r.Y)^\mathcal{I} \). Since \( \mathcal{I} \) is a model of the KB, \( (\exists r.Y)^\mathcal{I} \subseteq D^\mathcal{I} \). From this, it follows that \( C^\mathcal{I} \subseteq D^\mathcal{I} \).

**R6** Suppose \( C \) is added to \( U_n[D] \) via rule R6. Then \( D = \bot, Y \in U_{n-1}[\bot] \) for some \( Y \), and the axiom \( C \subseteq \exists r.Y \) is an axiom of the KB. By the inductive hypothesis, \( Y^\mathcal{I} \subseteq \bot^\mathcal{I} \) holds. Since \( \mathcal{I} \) is a model of the KB, it must be the case that \( C^\mathcal{I} = \emptyset \), and so clearly \( C^\mathcal{I} \subseteq D^\mathcal{I} \).
Now suppose that the application adds \((C, Y)\) to \(R_n[r]\) for some \(r\). Then one of the following rules must have been applied: R3, R7, R8. We take each in turn.

R3 \(C \in \mathit{U}_{n-1}[A]\) for some \(A\), and the rule \(A \subseteq \exists r.Y\) appears in the KB. By the inductive hypothesis, we have \(C^\mathcal{I} \subseteq A^\mathcal{I}\). Since \(\mathcal{I}\) is a model of the KB, \(A^\mathcal{I} \subseteq (\exists r.Y)^\mathcal{I}\). From this, \(C^\mathcal{I} \subseteq (\exists r.Y)^\mathcal{I}\).

R7 \((C, Y) \in R_{n-1}[s]\) and rule \(s \subseteq r\) appears in the KB. By inductive hypothesis, \(C^\mathcal{I} \subseteq (\exists s.Y)^\mathcal{I}\). Since \(\mathcal{I}\) is a model of the KB, \(C^\mathcal{I} \subseteq (\exists r.Y)^\mathcal{I}\).

R8 \((C, Z) \in R_{n-1}[s]\), \((Z, Y) \in R_{n-1}[t]\), and rule \(s \circ t \subseteq r\) appears in the KB. By inductive hypothesis, \(C^\mathcal{I} \subseteq (\exists s.Z)^\mathcal{I}\) and \(Z^\mathcal{I} \subseteq (\exists s.Y)^\mathcal{I}\). Since \(\mathcal{I}\) is a model of the KB, \(C^\mathcal{I} \subseteq (\exists r.Y)^\mathcal{I}\).

[Aside: Suppose \(x \in C^\mathcal{I}\). Then there exists a \(y\) such that \((x, y) \in s^\mathcal{I}\) and \(y \in Z^\mathcal{I}\). Also, there exists a \(w\) such that \((y, w) \in t^\mathcal{I}\) and \(w \in Y^\mathcal{I}\). Since \(\mathcal{I}\) is a model of the KB, \((x, w) \in r^\mathcal{I}\). Generalizing on \(x\), it follows that \(C^\mathcal{I} \subseteq (\exists r.Y)^\mathcal{I}\).]

A.3 Completeness

Given a knowledge base \(K\) in normal form, the completion \(\mathit{comp}(K)\) of \(K\) is obtained from \(K\) by exhaustively applying the completion rules from Table 5.1.

**Theorem A.3.1.** Let \(K\) be a knowledge base in normal form and let \(A, B\) be class names with \(K \models A \subseteq B\). Then \(A \subseteq B \in \mathit{comp}(K)\).
Proof. We define an interpretation $M$ of $\text{comp} K$ as follows.

$$\Delta^M = \{a_C \mid C \in BCO \cup \{\bot\}\} \quad \text{(the universe)}$$

$$A^M = \{a_C \mid C \sqsubseteq A \in \text{comp}(K)\}$$

$$R^M = \{(a_C, a_D) \mid C \sqsubseteq \exists r.D \in \text{comp}(K)\}$$

We next show that $M$ is a model of $\text{comp} K$, and thus also of $K \subseteq \text{comp}(K)$. So let $\alpha$ be any axiom in $\text{comp}(K)$.

If $\alpha$ is of the form $C \sqsubseteq A$ or $C \sqsubseteq \exists r.D$, then there is nothing to show.

For $\alpha$ of the form $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq D$, let $a_C \in C_1^M \cap \cdots \cap C_n^M$. Then $C \sqsubseteq C_1 \in \text{comp}(K)$ and $C \sqsubseteq C_n \in \text{comp}(K)$, i.e., $C \in U[C_1], C \in U[C_2], \ldots, C \in U[C_n]$. Thus $C \in U[D]$ i.e., $C \sqsubseteq D \in \text{comp}(K)$ by rule R2. Hence $a_C \in D^M$ as required.

For $\alpha$ of the form $\exists r.Y \sqsubseteq B$, let $(a_C, a_D) \in R^M$ and $a_D \in A^M$. Then $(X, Y) \in R[r]$. By R5, we then have $X \in U[B]$ i.e., $X \sqsubseteq B \in \text{comp}(K)$ and thus $a_C \in B^M$ as required.

For $\alpha$ of the form $X \sqsubseteq \exists r.Y$, let $(a_C, a_D) \in R^M$. Then $Y \sqsubseteq \bot \in \text{comp}(K)$. Since $Y \sqsubseteq \bot, X \sqsubseteq \bot$ i.e., $X \in U[\bot]$.

For $\alpha$ of the form $r \subseteq s$ let $(a_C, a_D) \in R^M$. Then $C \sqsubseteq \exists r.D \in \text{comp}(K)$ and by rule R7 we obtain $C \sqsubseteq \exists s.D \in \text{comp}(K)$, and thus $(a_C, a_D) \in S^M$ as required.

For $\alpha$ of the form $r_1 \circ r_2 \subseteq r$ let $(a_C, a_D) \in R_1^M$ and $(a_D, a_E) \in R_2^M$. Then $C \sqsubseteq \exists r_1.D \in \text{comp}(K)$ and $D \sqsubseteq \exists r_2.E \in \text{comp}(K)$, and by rule R8 we obtain $C \sqsubseteq \exists r.E \in \text{comp}(K)$, and thus $(a_C, a_E) \in R^M$ as required.

Now assume $A \sqsubseteq B \not\in \text{comp}(K)$. Then there is some $a_C \in A^M$ with $C \sqsubseteq A \in \text{comp}(K)$ and $a_C \not\in B^M$. We obtain $a_C \in A^M \setminus B^M$ and thus $M \not\models A \sqsubseteq B$ and
$K \nsubseteq A \subseteq B$. The claim is thus proven by contraposition.
Bibliography


In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*,
SOSP ’03, pages 29–43, New York, NY, USA, 2003. ACM.

[31] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT:

PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Pro-
ceedings of the 10th USENIX Conference on Operating Systems Design and Imple-

[33] Eric L. Goodman, Edward Jimenez, David Mizell, Sinan al Saffar, Bob Adolf, and
David Haglin. High-performance Computing Applied to Semantic Databases. In *Pro-
ceedings of the 8th Extended Semantic Web Conference on The Semantic Web: Research and Applications - Volume Part II*, ESWC’11, pages 31–45, Berlin, Hei-

[34] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description
Logic Programs: Combining Logic Programs with Description Logic. In Gusztáv
Hencsey, Bebo White, Yih-Farn Robin Chen, László Kovács, and Steve Lawrence,

Cichlid: Efficient Large Scale RDFS/OWL Reasoning with Spark. In *2015 IEEE
International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyder-


[38] Steve Harris and Andy Seaborne, editors. SPARQL 1.1 Query Language. W3C Recommendation.


[44] Aidan Hogan, Jeff Z. Pan, Axel Polleres, and Stefan Decker. SAOR: Template Rule Optimisations for Distributed Reasoning over 1 Billion Linked Data Triples. In Pe-


