Rule Mining on Distributed RDF Data

Kunal Jha

Matriculation number: 2973292

September 5, 2018

Master Thesis

Computer Science

Supervisors:

Prof. Dr. Axel-C. Ngonga Ngomo, Tommaso Soru,
Dr. Hajira Jabeen, Gezim Sejdiu, Prof. Dr. Jens Lehmann.

Institute for Informatik

Rheinische Friedrich-Wilhelms-Universität Bonn

Declaration of Authorship

I, Kunal Jha, declare that this thesis, titled "Rule Mining on Distributed RDF Data", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my own work. I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

Acknowledgements

This work would not have been possible without the guidance and support of Prof. Dr. Axel-C. Ngonga Ngomo of the The Data Science (DICE) Group at University of Paderborn, Dr. Hajira Jabeen and Prof. Dr. Jens Lehmann of the Smart Data Analytics (SDA) Group at University of Bonn. They have been supportive of my career goals and worked actively to provide me with their academic time to pursue those goals. They consistently allowed this thesis to be my own work, but steered me in the right direction whenever they thought I needed it.

I am grateful to Tommaso Soru for his active support in the algorithm development and optimization. I would like to thank Gezim Sejdiu and Ivan Ermilov for their constant support with technical plateaus and help me attain the maximum possible optimization in terms of Spark on the available cluster. Without their valuable participation and support into building the idea and technology stack, the project would not have been successfully completed. Finally, I would like to thank Michael Röder for his valuable inputs in planning and ideas. I am also gratefully indebted to all of them for their valuable comments on this thesis. I would also like to thank all the reviewers for taking out time to review the rules and participate in the survey.

Nobody has been more important to me in the pursuit of this project than the members of my family. I would like to thank my parents, whose love and guidance are with me in whatever I pursue. Finally, I thank with love to my partner, Nauf, who has been my best friend and great companion, loved, supported, encouraged, entertained, and helped me get through this period in the most positive way. Thank you.

Contents

List of Figures

Abstract

Over the recent years, the advances in information extraction has resulted in the proliferation of large Knowledge Bases (KBs), which is a machine readable collection of knowledge. The KBs are inevitably bound to be incomplete. Logic rules are being used to predict new facts based on the existing ones. Horn rules are particular rule-like form which gives it useful properties for use in logic programming and are one of the popular form of rules mined. However, Horn rules do not take into account possible exceptions, so that predicting facts via such rules introduces errors. Another major challenge is to perform scalable analysis of large scale KBs to facilitate rule mining. Hence we address these two problems in this work and present two rule mining approaches implemented for a cluster based environment - (1) Exception Enriched Rule Mining Approach, which aims at adding exception (negated body elements) in order to refine rules; (2) Decision Tree Rule Mining, a rule miner which can mine both horn rules and non-monotonic rules. Both these approaches are implemented for a distributed environment. We apply our method to discover rules with exceptions from real-world KBs. We test them on KBs upto 10 million triples and have achieved good results for the same.

Keywords: RDF, Rule mining, large-scale

Chapter 1

Introduction

Over the recent years, the advances in information extraction (IE) has resulted in the proliferation of large Knowledge Bases (KBs), also referred as Knowledge Graphs (KGs). KBs are a machine readable collection of knowledge. The KGs we focus on in this work is in the Resource Description Format (RDF) and hence, relational facts are in the form of subject-predicate-object (SPO) triples. These KBs have primarily been constructed by mining the Web for information resulting in their size increasing to be large enough that they can themselves be mined for information.

1.1 Motivation and Problem

Despite the prodigious advances in IE and these knowledge bases spanning manifold domains, these KBs are usually far from being complete. Hence to increase the number of facts, logic rules are being used based on the existing ones. Therefore, learning reliable rules from knowledge bases becomes increasingly paramount. The rules mined primarily serve two fold purpose. Firstly, since KBs operate under the Open World Assumption (OWA) (i.e., missing facts are treated as unknown instead of being necessarily false), the rules can be used to derive new facts which in turn make the KB more complete. Secondly, the presence of rules for a KB also serves as a check for consistency between facts and hence, remove erroneous facts. Additionally, these rules carry human intelligible observations. For example, in the following rule, the fact that living is detrimental factor in deciding where a person lives has a value in itself [1].

$$bornIn(x, y) \wedge citizenOf(x, z) \implies cityOf(y, z)$$

To complete and curate a KG, inductive logic programming (ILP) and data mining techniques (e.g., [2],[3] , [4]) have been used to identify prominent patterns and cast them in the form of Horn rules. The starting point of most ILP and rule mining approaches are Horn rules. For example, consider the following rule, *"Someone who works at Uni Bonn and teaches is a professor at Uni Bonn."*,

$$worksAtUniBonn(X) \wedge teachesClass(X) \implies profAtUniBonn(X)$$

However, in this case, both PhD. students and Professors are a part of the facts and for all facts where X is a PhD. student the rule has an exception. These exceptions are discussed in the work by Gad-Elrab, Mohamed H., et al.,[5]. It proposes methodologies to mine rules with a negated body element to catch such exceptions. In our example, the rule is altered as follows, *"Someone who works at Uni Bonn, teaches and is not doing PhD is a professor at Uni Bonn"*,

$$\begin{aligned} &worksAtUniBonn(X) \wedge teachesClass(X) \wedge \\ &\qquad \neg isAPhDStudent(X) \implies profAtUniBonn(X) \end{aligned}$$

This additional knowledge (i.e., the predicate, *doingPhD*) gives a better explanation of outlier facts for the original rule.

The second dimension to the rule mining task is the explosive size of data. In order to deal with this explosion, models of cluster computing, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing, became popular. Implementations of this model in systems like Map Reduce [6], Dryad [7] and Map-Reduce Merge [8] achieve the scalability and fault tolerance by providing a programming model where the user can create an acyclic data flow graph to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention. The Hadoop Map Reduce, is the most popular open source implementation of the MapReduce framework proposed by Google [6].

In the recent years, a new cluster computing framework called Spark [9] has gained significant popularity as it supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce. In comparison to the Hadoop MapReduce, Spark framework has proven to be give better performances in tasks like fast data processing, iterative processing, graph processing, machine learning and joining datasets. These tasks used in different combinations form a major number of rule mining algorithms and hence, the algorithms can be extended on Spark in order to re-implement the algorithms on the "big" KBs.

1.2 Objectives and Contributions

The primary objective of this work is to mine Horn rules with positive and negative body elements (i.e., exceptions) from RDF Knowledge Bases in a distributed cluster based environment using Spark. This objective is implemented in two different approaches. Firstly, we present the re-implementation of exception rule mining [5] in a distributed environment. It takes a KG and a set of Horn rules as input and yields a set of exception-enriched rules as output. The output rules are no longer necessarily Horn clauses. In this approach, we tailor the original algorithm to compute what is called exception witnesses: predicates that are potentially involved in explaining exceptions. We generate nonmonotonic rule candidates that we could possibly add to our KG rules. Secondly, we present a distributed cluster based novel approach to mine rules. The approach the computation and performance advances of Spark in order to build decision trees on top of the KB and mine rules with both positive body elements as well as exceptions. We evaluate both these methodologies based on runtime and quality of rules (based on human evaluation). The source code of the project is available at `https://github.com/dice-group/LINDA`.

More precisely, the contributions of this work are as follows:

1. A Spark cluster based distributed framework for nonmonotonic rule mining as a knowledge revision task, to capture exceptions from Horn rules by using the definitions as originally proposed in Gad-Elrab, Mohamed H., et al.,[5], for a distributed environment.

2. A novel Spark cluster based distributed approach of rule mining using Spark Machine Learning library (Decision Tree based) to mine rules with both positive and negative bodies i.e., combining the outcomes of all the previous works.

3. The rule revision task shows great run times as well as rule quality in human evaluation as well as link prediction. There are some scope of improvements due to limitation of Spark which are addressed in Future works.

4. The decision tree based approach was unfortunately not tested in the same environment due to some technical networking issues in the cluster setup. We present the result for a standalone system which look promising in terms of run time and quality. However, the approach does have some serious memory concerns when it comes to transforming the data.

1.3 Thesis Structure

The remainder of this document is structured as follows. The Related Work chapter discusses the related works and Preliminaries chapter introduces all the required definitions and implementation elements required to better understand the work. The Learning Exception-enriched Rules chapter recaptures the exception witness set from the work [5] and discusses the algorithm and implementation for a distributed environment. Rule Mining using Decision Tree in Spark introduces the Decision Tree based approach and its implementation details. Chapter Evaluation presents the experiments and results before the chapter Conclusions and Future Work, concludes .

Chapter 2

Related Work

In this chapter, we focus on previous work that is related to the problem stated in Introduction. Our main goal is to mine rules in a distributed environment. These rules contains either positive, negative body atoms or both. Rule has been an area of active research during the past years. Approaches for predicting unseen data in KGs can be roughly divided into statistics-based, association-based and logic-based. Statistical Relational Learning is concerned with the creation of statistical models for relational data. They apply techniques like tensor factorization model, latent feature models, neural embedding based models [10]. The association rules [11] are mined on a list of items known as transactions. In these works, some works ([5, 12]) concentrated on finding exception rules . The logical based focuses more on logical rule learning [3, 4]. The work [13] originally proposed rule mining using association rule mining, enabled by concept of mining configurations. They presented rule-based approaches for predicate suggestion, data enrichment, ontology improvement, and query relaxation. They aimed at prevent inconsistencies in the data through predicate suggestion, enrichment with missing facts, and alignment of the corresponding ontology.

Jozefowska et al. [14] proposes an algorithm for frequent pattern mining in KBs that uses DL-safe rules. Such KBs can be transformed into a disjunctive datalog program, which allows seeing patterns as queries. Some approaches use rule mining for ontology merging and alignment. In [15], association rules and frequency analysis are used to identify and classify common misusage patterns for relations in DBpedia. In the same fashion, Abedjan and Naumann [16] applies association rules to find synonym predicates in DBpedia. The matched synonyms are then used for predicate expansion in the spirit of data integration.

Since RDF data can be seen as a graph, mining frequent subtrees [17, 18] is another related field of research. [3] focuses on mining frequent predicate cycles on the RDF Graph. RDF2Rules first mines a kind of interesting frequent patterns in KBs, which are called Frequent Predicate Cycles (FPCs); then multiple rules are generated from each mined FPC. It uses entity type information when generating and evaluating learned rules, which results in rules having more accurate predictions.

Among the frameworks based on Markov Logic Networks, majority do not provide the possibility of discovering rules. The works like NetKit-SRL[19], ProbCog [20] need rules as input in order to proceed with the weight learning, grounding, and inference phases. Alchemy [21] instead performs the construction of rules in a combinatorial way; however, this method is highly inefficient, especially on datasets having an elevated number of distinct properties. Even the Typical ILP systems, such as FOIL [22] and Progol [23], need a set of training examples that contain both positive and negative examples of the target concepts or relations. Although ILP systems such as [24] are proposed to learn rules from only positive examples, the main problem with these approaches is the low efficiency when dealing with large KBs.

Sherlock [25] is an unsupervised ILP method to learn first-order Horn clauses from open domain facts. It uses probabilistic graphical models (PGMs) to infer new facts. It tackles the noise of the extracted facts by extensive filtering and penalizes longer rules in the inference part. It uses statistical significance and relevance to mine rules. AMIE [26] mainly focuses on evaluation methodologies of rules under the Open World Assumption(OWA), and its searching strategy becomes inefficient when dealing with large-scale KBs and long rules. It learns one rule at a time by gradually adding new atoms to the rule body. It is based on a formal model for rule mining under the open-world assumption, a method to simulate counterexamples, and a scalable mining algorithm. Recently, AMIE has been extended to AMIE+[4] by a series of pruning and query rewriting techniques, both lossless and approximate which makes it more efficient in terms of runtime, number and quality of output rules. AMIE and AMIE+, use only declared statements as counterexamples, thus embracing the open-world assumption (OWA).

`HornConcerto` [27] is a SPARQL based implementation for mining rules in large directed labelled graphs. It is a complete algorithm i.e., it runs without constraints, and can detect all Horn rules of a given type. It outperforms AMIE in terms of runtime and memory consumption, while scaling to very large knowledge bases with high quality rule in datasets having hundreds of millions of triples without the need of a schema and achieve state-of-the-art link predictions on a widely-used benchmark. It suggests link prediction as an evaluation methodology rule mining task which we use for this work along with human evaluation of the mined rules.

However, these works focused primarily on mining Horn Rules. The work

[12] concentrated on finding (interesting) exception rules, which are defined as rules with low support (rare) and high confidence. However, the work focused primarily on rare rules rather than exception. The work [5] addresses the problem of finding exception and is the major inspiration for this work. It presents a method for mining non-monotonic rules from KGs: first learning a set of Horn rules, and then revising them by adding negated atoms into their bodies with the goal of improving the confidence of a rule set for data prediction. It introduces a quality function in order to select the best revision from potential candidates we devised rule-set ranking measures, based on data mining measures and the novel concept of partial materialization.

In contrast to the aforementioned approaches, our approach is implemented on Spark which is a unified analytics engine for large-scale data processing. We use the idea proposed by the work [5] and implement a distributed cluster based Exception Enriched Rule mining system. We use the Spark Dataframes and its processing optimization to achieve the results via series of joins rather in contrast to the original algorithm in a much faster distributed environment. We also ensure the usage of algorithm to be horizontally scaled i.e., essentially the addition of more machines or setting up a cluster or a distributed environment. We also propose a decision tree based approach that combine the mining of both horn and non monotonic rules exploiting the distributed processing power of Spark. Both our approaches have a faster run time than the state of the art approaches (some of them mentioned in this section). We also use link prediction for quality evaluation of the rules along with human evaluation results.

Chapter 3

Preliminaries

3.1 Semantic Web

The term *Semantics* comes from the branch of philosophy called *semiotics*, which is primarily focused around study of meaning. It explains how we derive meaning from words or symbols. Semantic Data is the data organized in a format that it's readily understandable by machines without human intervention. The fields like Natural Language Processing and Machine Learning when combined with semantics allows us to see and discover relationships and data. The field of semantic web presents a set of methods or technology used that enable the machine to understand the meaning / semantics of content or information on the web. It is to be noted that semantic web is not a separate web but an extension of the World Wide Web (WWW). It is an effort to enhance current web so that machines can process the information presented on WWW, interpret and connect it, to help us to find required knowledge. Similar to WWW being a huge distributed hypertext system, semantic web is intended to form a huge distributed knowledge based system, however, the focus of semantic web is to share data instead of documents. In other words, it is a project that aims provide a common framework allowing data to be shared and reused across application, enterprise, and community boundaries. The figure 3.1 shows the architecture of Semantic Web.

Uniform Resource Identifier (URI)

The first layer, URI and Unicode, are taken from the WWW. Unicode is a standard of encoding international character sets and allows human languages to be used on the web using one standardized form. Uniform Resource Identifier (URI) is a string of a standardized form that allows to uniquely identify resources across the web. A subset of URI is Uniform Resource Lo-

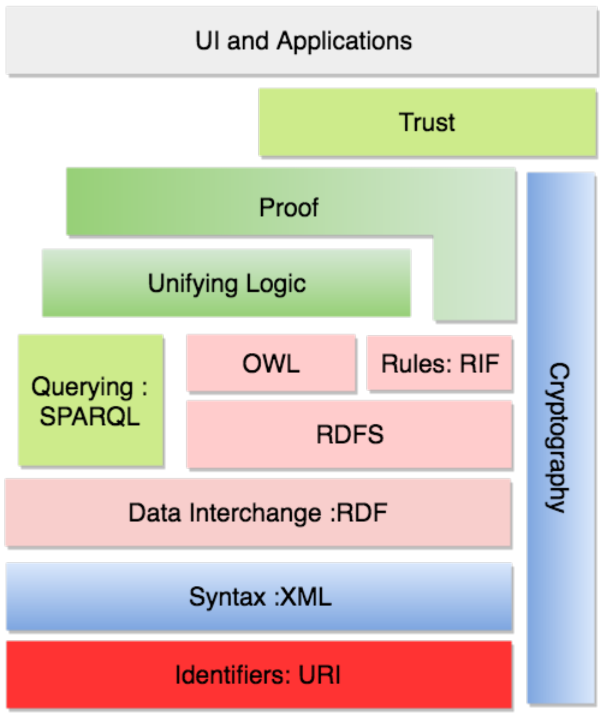

Figure 3.1: Semantic Web Stack

cator (URL), which contains access mechanism and a network location of a document. Another subset of URI is URN that allows to identify a resource without implying its location and means of de-referencing it. The usage of URI is important for a distributed internet system as it provides understandable identification of all resources.

XML

The next is the Extensible Markup Language (XML) layer with XML namespace and XML schema definitions. It makes sure that there is a common syntax used in the semantic web. XML is a general purpose markup language for documents containing structured information. A XML document contains elements that can be nested and that may have attributes and content. XML namespaces allow to specify different markup vocabularies in one XML document. The code representation of RDF in XML format is follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns:dc="http://purl.org/dc/elements/1.1/"
```

```
        xmlns:region="http://www.random.fake/">

        <rdf:Description rdf:about="http://en.wikipedia.org
        /wiki/Bonn">
                <dc:title>Bonn</dc:title>
                <dc:state>North Rhine-Westphalia</dc:state>
                <dc:publisher>Wikipedia</dc:publisher>
                <region:population>300,000</region:population>
                <region:principaltown rdf:resource="http://www.
                random.fake/bonn"/>
        </rdf:Description>

</rdf:RDF>
```

RDF

The universal framework of semantics is called the Resource Description Framework (RDF) [28]. It is an infrastructure that enables the encoding, exchange and reuse of structured metadata. It uses XML as a common syntax and provides a model for describing resources. It is a standard for modelling data and is structured while being simple, flexible, distributed and decentralized. It is a framework for representing information about resources in a graph form. It was primarily intended for representing metadata about WWW resources, such as the title, author, and modification date of a Web page, but it can be used for storing any other data. It is based on triples subject-predicate-object that form graph of data. All data in the semantic web use RDF as the primary representation language. RDF itself serves as a description of a graph formed by triples. RDF has something known as *resources* as any object that is uniquely identifiable by URI (Uniform Resource Identifier). Resources have properties (attributes or characteristics), which are identified by property types. These properties define a unidirectional relationship of a resource with values, which is atomic in nature (string, text, numbers, etc.), or another resource that may have its own properties. A collection of these properties which refers to the same resource is called a description. Figure 3.2 depicts an RDF graph. The triple is the basic unit of rdf and is defined as follows:

- **Subject**: Subject represents a resource which the triple presents the information about. The subject usually is comprised of an URI or an empty node.

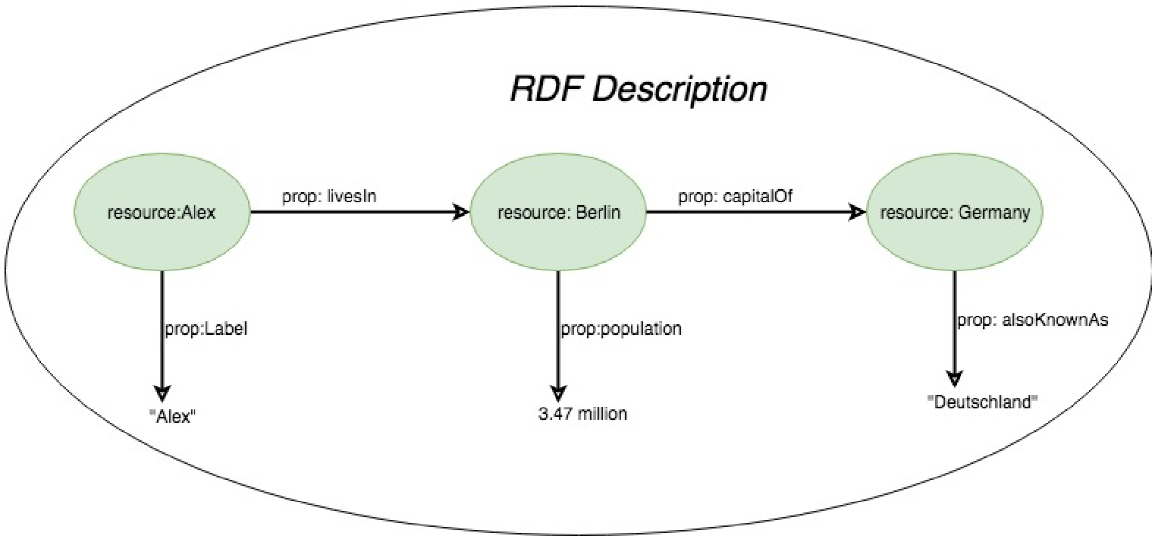

Figure 3.2: RDF Model

- **Predicate:** Predicate defines exactly what kind of information the triple will express for the subject. The predicate should contain, an URI.
- **Object:** The object defines the value expressed by the predicate. The object can be either an URI, a label (plain text) or an empty node.

RDFS

In order to attain standardized description of taxonomies and other ontological constructs, a RDF Schema (RDFS) [29] was created together with its formal semantics within RDF. RDF Knowledge bases contain RDFS schema which is used to declare vocabularies, the sets of semantics property types defined by a particular community. The schema defines the valid properties in RDF description, as well as any characterstics or restrictions of the property types values themselves. RDFS can be used to describe taxonomies of classes and properties and use them to create lightweight ontologies. Some of the classes are as follows :

- `rdfs:Resource` This is the class of everything. All things described by RDF are resources.
- `rdfs:Class` This is the class of resources that are RDF classes.

- `rdfs:Literal` This is the class of lateral values such as the character list and numbers.
- `rdf:XMLLiteral` – This is the class of XML literal values.
- `rdfs:Property` This is the class of RDF properties. These are used to define relationship in ontologies.

Properties are instances of the class `rdfs:Property` and describe a relation between subject resources and object resources. Some of the properties are :

- `rdf:type` It connects a resource to its class. In other words, this feature is used to declare a resource to be an instance of the given class.
- `rdfs:subClassOf` It connects to a given class with its over-class. All instances of the class are also instances of over-class. A class may be sub-class of more than one class.
- `rdfs:domain` Specifies the domain of the property, which is the class of resources that may be subject to the queues containing this predicate. If this domain is not specified then any resource is pre-defined as a subject.
- `rdfs:range` Specifies the rank of a property, which is the class of these resources that can be objects in a triad containing this predicate.
- `rdfs:subPropertyOf` is an instance of rdf:Property that is used to state that all resources related by one property are also related by another.
- `rdfs:label` is an instance of rdf:Property that may be used to provide a human-readable version of a resource's name.
- `rdfs:comment` is an instance of rdf:Property that may be used to provide a human-readable description of a resource.

OWL

The Web Ontological Language (OWL) [30] is a language derived from description logic, and offers more constructs over RDFS. It is syntactically embedded into RDF, so similar to RDFS, it provides additional standardized vocabulary. Since OWL is based on description logic, it has a formal semantics defined.RDFS and OWL have semantics defined and this semantics can be used for reasoning within ontologies and knowledge bases described using these languages. OWL has three variant sub languages -

- `OWL Lite:` It is the syntactically simplest sub-language. It is intended to be used in situations where only a simple class hierarchy and simple constraints are needed.

- `OWL-DL:` It is much more expressive than OWL-Lite and is based on Description Logic (hence the suffix DL). Description Logic are a decidable fragment of First Order Logic and are therefore amenable to automated reasoning. It is therefore possible to automatically compute the classification hierarchy and check for inconsistencies in an ontology that conforms to OWL-DL.

- `OWL-Full:` It is the most expressive OWL sub-language. It is intended to be used in situations where very high expressiveness is more important than being able to guarantee the decidability or computational completeness of the language. It is therefore not possible to perform automated reasoning on OWL-Full ontologies.

A sample owl file as defined in XML looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns:owl="http://www.w3.org/2002/07/owl#"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

 <owl:Class rdf:ID="King">
    <rdfs:subClassOf rdf:resource= "http://xmlns.com/foaf/0.1
    /Person"/>
       <owl:equivalentClass>
          <rdf:Description rdf:about= "http://www.vocab.org/
          ontology/portal#King">
          <owl:equivalentClass rdf:resource="#King"/>
          </rdf:Description>
    </owl:equivalentClass>
 </owl:Class>
</rdf:RDF>
```

SPARQL

For querying RDF data as well as RDFS and OWL ontologies with knowledge bases, a Simple Protocol and RDF Query Language (SPARQL) is available.

SPARQL is SQL-like language, but uses RDF triples and resources for both matching part of the query and for returning results of the query. Since both RDFS and OWL are built on RDF, SPARQL can be used for querying ontologies and knowledge bases directly as well. SPARQL is not only query language, it is also a protocol for accessing RDF data.

It is expected that all the semantics and rules will be executed at the layers below proof and the result will be used to prove deductions. Formal proof together with trusted inputs for the proof will mean that the results can be trusted, which is shown in the top layer of the figure above. For reliable inputs, cryptography means are to be used, such as digital signatures for verification of the origin of the sources. On top of these layers, application with user interface can be built.

3.2 RDF Knowledge Graphs

On the Web, knowledge graphs (KG) are usually encoded using the RDF data model [28], which represents the content of the graph with statements which is a set of triples. A triple is a mathematical structure that is uniquely defined by its three components. An RDF statement i.e., a triple is a fact and hence within a KG, these triples encode positive facts about the world, and they are naturally treated under the Open World Assumption(OWA).

Definition 3.2.1. RDF Knowledge Graph An RDF knowledge graph $G = (V, E)$ is modeled as a set of triples $(s, p, o) \in (R \cup B) \times P \times (R \cup B \cup L)$. where $R, B, L \in V$ and $P \in E$ and are defined as,

- R is the set of all RDF resources, which stand for things of relevance in the domain to model.
- B is the set of all RDF blank nodes, i.e., which describe existential quantification of entities that do not need a unique global identifier
- $P \subseteq R$ is the set of all RDF predicates and stands for the sets of binary relations which can exist between resources or literals.
- L is the set of all literals, i.e., of all data values used in a given knowledge graph

One key element of knowledge base is given by the operations used to build the terminology and these operations are directly related to the forms and the meaning of the declarations allowed in the TBox. The TBox contains intentional knowledge in the form of a terminology and is built through declarations that describe general properties of concepts.

Definition 3.2.2. TBox A TBox T is a finite collection of concept inclusion axioms of the form $C \subseteq D$ and concept equivalence axioms of the form $C \equiv D$, where C and D are concepts.

In this work we consider a KG without blank nodes or schema.

3.3 Rule Mining

Rule mining [31] can be defined as,

Definition 3.3.1. *Given a set of training examples, find a set of rules that can be used for prediction or classification of new instances.* More accurately, we aim to find a hypothesis as a set of rules described in the hypothesis description language, providing the definition of the target concept which is

- complete, i.e., it covers all examples that belong to the concept, and
- consistent, i.e., it does not cover any example that does not belong to the concept.

Majorly, we come across two kinds of rule learning- Concept Learning and Propositional Rule Learning. For the Concept learning, task is to learn a set of rules that describe a single target class, also called the target concept. As training information, we are given a set of positive examples, for which we know that they belong to the target concept, and a set of negative examples, for which we know that they do not belong to the concept. In this case, it is typically sufficient to learn a theory for the target class only. All instances that are not covered by any of the learned rules will be classified as negative.

In case of Propositional rules a classification rule is an expression of the form: `IF Conditions THEN C` where `C` is the class label, and the Conditions are a conjunction of simple logical tests describing the properties of instances that have to be satisfied for the rule to infer `C`. Thus, a rule essentially corresponds to an implication `Conditions` $\longrightarrow$ `C` in propositional logic.

In real world attaining the goals of completeness and consistency are unrealistic in learning from large, noisy datasets, which contain random errors in the data, either due to incorrect class labels or errors in instance descriptions. Forcing the algorithm to learn a complete and consistent hypothesis is undesirable in the presence of noise, because the hypothesis tries to explain the errors as well. This is known as overfitting the data. In order to not overfit but still attain a fairly valid hypothesis, the consistency and completeness requirements need to be relaxed and replaced with some other evaluation criteria, such as sufficient coverage of positive examples, high predictive accuracy of the hypothesis or its significance above the requested, predefined threshold. These measures can be used both as heuristics to guide rule construction and as measures to evaluate the quality of induced hypotheses.

3.4 Horn Clause

A Horn clause in case of rule mining in RDF KB can be defined as an implication from a disjunction of antecedant (a set of atoms) to a consequent (a single atom) where for all the variables in the consequent must occur in at least one of the antecedant. In the context of first-order logic, an atom is a formula which contains a predicate and a list of undefined arguments and cannot be divided into sub-formulas. The typical structure of these rules expects an implication sign, an atom h as the head (i.e., the implication target), and a set of atoms $b_1, b_2, ...b_n$ as the body. The body can be composed by one or more conjunction literals. More concisely in case of RDF rule mining,

$$\bigwedge_{i=1}^{n} b_i \implies h \tag{3.1}$$

where $B = \{b_1, b_1, b_2...b_n\}$ are positive body predicates (atoms) and $H = \{h\}$ is the head predicate(atom) , and the predicate is the P from the RDF triple. For the ease of simplicity and following from the previous works, we focus our work on unary predicates. We translate the binary relation into multiple unary relations by concatenating the predicate with one of its arguments (objects). For example,

$$worksAtUniBonn(X) \wedge teachesClass(X) \Rightarrow profAtUniBonn(X)$$

3.5 Nonmonotonic Logic Programs

The term “non-monotonic logic” covers a family of formal frameworks devised to capture and represent de-feasible inference, i.e., that kind of inference of everyday life in which reasoners draw conclusions tentatively, reserving the right to retract them in the light of further information. Such inferences are called “non-monotonic” because the set of conclusions warranted on the basis of a given knowledge base does not increase (in fact, it can shrink) with the size of the knowledge base itself. A logic is non-monotonic if some conclusions can be invalidated by adding more knowledge. The logic of definite clauses with negation as failure is non-monotonic. Non-monotonic reasoning is useful for representing defaults. Mathematically, a nonmonotonic logic program [32] P is a set of rules of the form

$$\bigwedge_{i=1}^{n} b_i \wedge \bigwedge_{i=1}^{m} \neg e_i \Longrightarrow h \tag{3.2}$$

where $B = \{b_1, b_1, b_2 ... b_n\}$,are positive body predicates , where $E = \{e_1, e_1, e_2$ $...e_m\}$ are exceptions and combine to form the negated part of the body and $H = \{h\}$ is the head.

3.6 Standard confidence

The support of a Horn clause estimates the probability of the rule head to hold true, given the body:

$$P(H|B) = \frac{P(H \cap B)}{P(B)} \tag{3.3}$$

The standard confidence of a rule $R := B \Rightarrow H$ is defined as :

$$c(R) = \frac{\{H \wedge B\}}{B} \tag{3.4}$$

3.7 Spark

Spark[9] is an open source cluster computing framework for batch and real-time processing. It is built on top of Hadoop MapReduce model and extends it in order to use it for more types of computations, including interactive queries and stream processing, in a more efficient manner. It provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance. The major advantages of using Spark are as follows:

- It runs up to 100 times faster than Hadoop MapReduce for large -scale data processing in memory and 10 times faster when running on disk via controlled partitioning. It manages data using partitions that help parallelize distributed data processing with minimal network traffic.
- It supports multiple data sources such as Parquet, JSON, Hive and Cassandra apart from the usual formats such as text files, CSV and RDBMS tables.

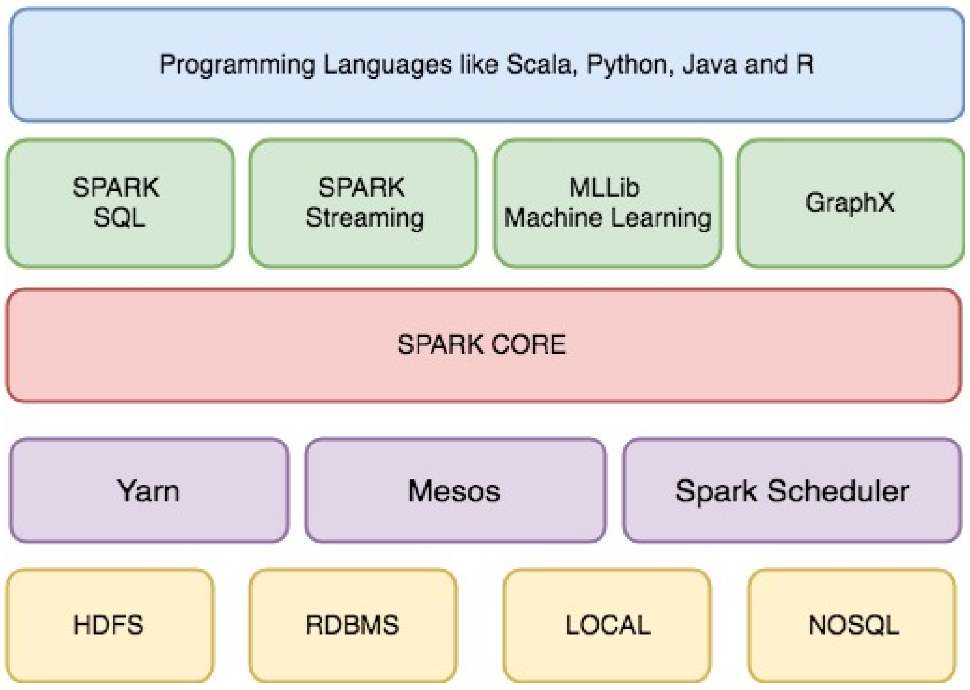

Figure 3.3: Spark Framework

- It delays its evaluation until it is absolutely necessary which is one of the key factors contributing to its speed. For transformations, Spark adds them to a DAG (Directed Acyclic Graph) of computation and only when the driver requests some data, does this DAG actually gets executed.
- Its computation is real-time and has low latency because of its in-memory computation and is designed for massive scalability.
- Its machine learning component comes in handy when it comes to big data processing eradicating the need to use multiple tools, for processing and learning separately.
- Finally, it provides a smooth compatibility with Hadoop and has the ability to run on top of an existing Hadoop cluster using YARN for resource scheduling thus, eliminating the need to replace existing clusters.

The operations on RDDs (basic unit of Spark), are mainly classified into two categories: transformations and actions. With the operations of transformations, the user can create a new dataset from an existing RDD. After the operation of actions, a value is returned to the driver program. When a job is committed to the master of the cluster, a DAG is built from the RDD's lineage graph. A DAG consists of several stages. The stages are divided into two categories: shuffle map stage and result stage. Shuffle map stages are those that their results are input for another stage, while result stages are those that their tasks directly compute the action that initiated a job (count, collect, save, etc.).The figure 3.3 gives an overview of Spark's framework.

3.7.1 Spark Resource Manager

The lowermost layer of Spark is the data layer which can be databases like RDBMS and NoSQL, or even machine's file system in case of local standalone setup. In this work, the dataset is hosted on HDFS (Hadoop Distributed File System) . It is a Java based file system that provides scalable and reliable data storage, and is designed to span large clusters of commodity servers. It has a master/slave architecture. A HDFS cluster [33] consists of a single NameNode, a master server responsible for managing the file system and regulates access to files by clients, and a number of DataNodes, usually one per node in the cluster, manage storage attached to the nodes that they run on. It stores each file as a sequence of blocks. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file.The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Block report from each of the Data Nodes in the cluster.

As mentioned, Spark applications are run as independent sets of processes on a cluster, all coordinated by a central coordinator (the purple layer in 3.3). This central coordinator can connect with three different cluster managers, Spark's Standalone, Apache Mesos, and Hadoop YARN (Yet Another Resource Negotiator) [34]. The `Spark Standalone` cluster manager is a simple cluster manager available as part of the Spark distribution. It has high availability for the master and can manage resources per application while alongside of an existing Hadoop deployment and access HDFS data. It uses a simple FIFO scheduler for applications and each application uses all the available nodes in the cluster in the default mode.The Spark standalone cluster manager supports automatic recovery of the master by using standby masters using ZooKeeper. It also supports manual recovery using the file system. The cluster is resilient to Worker failures regardless of whether recovery of the master is enabled. It also has a Web UI to monitor the application allowing monitoring of information about tasks running in the application, executors, and storage usage. `Apache Mesos` is a distributed systems kernel having a high availability for masters and slaves, with the ability to manage resources per application, and also support for Docker containers. The master makes offers of resources to the application which either accepts the offer or not. Thus, claiming available resources and running jobs is determined by the application itself. This cluster supports automatic recovery of the master using ZooKeeper to enable recovery of the Master. Finally, it provides numerous metrics,like percentage and number of allocated cpu's, memory used, percentage of available memory used, disk space etc., for the master and slave nodes accessible via a URL. `Hadoop YARN` is a distributed computing framework for job scheduling and cluster resource management, with high availability for masters and slaves along with a pluggable scheduler.It has a ResourceManager with two parts, a *Scheduler*, and an *ApplicationsManager*. The Scheduler is a pluggable component. The Scheduler has two popular implementation - a CapacityScheduler, useful in a cluster shared by more than one organization, and the FairScheduler, which ensures all applications, on average, get an equal number of resources. Both schedulers assign applications to a queues and each queue gets resources that are shared equally between them. Within a queue, resources are shared between the applications. The ApplicationsManager is responsible for accepting job submissions and starting the application specific ApplicationsMaster. Finally, it also a Web UI for the ResourceManager and the NodeManager. The ResourceManager UI provides metrics for the cluster while the NodeManager provides information for each node and the applications and containers running on the node.

3.7.2 Spark Data Source

A Spark RDD [9] (resilient distributed dataset) as a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage i.e., it can always be reconstructed in the event of failures. They are lazy and ephemeral i.e., partitions of a dataset are materialized on demand when they are used in a parallel operation. A DataFrame [35] is a distributed collection of rows with a homogeneous schema. They are collections of structured records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. It is conceptually equivalent to a table in a relational database with richer optimization under the hood. They can be created directly from Spark's built-in distributed collection, enabling relational processing in existing Spark programs. Designed to make large data sets processing easier, it allows imposition of a structure onto a distributed collection of data, al-

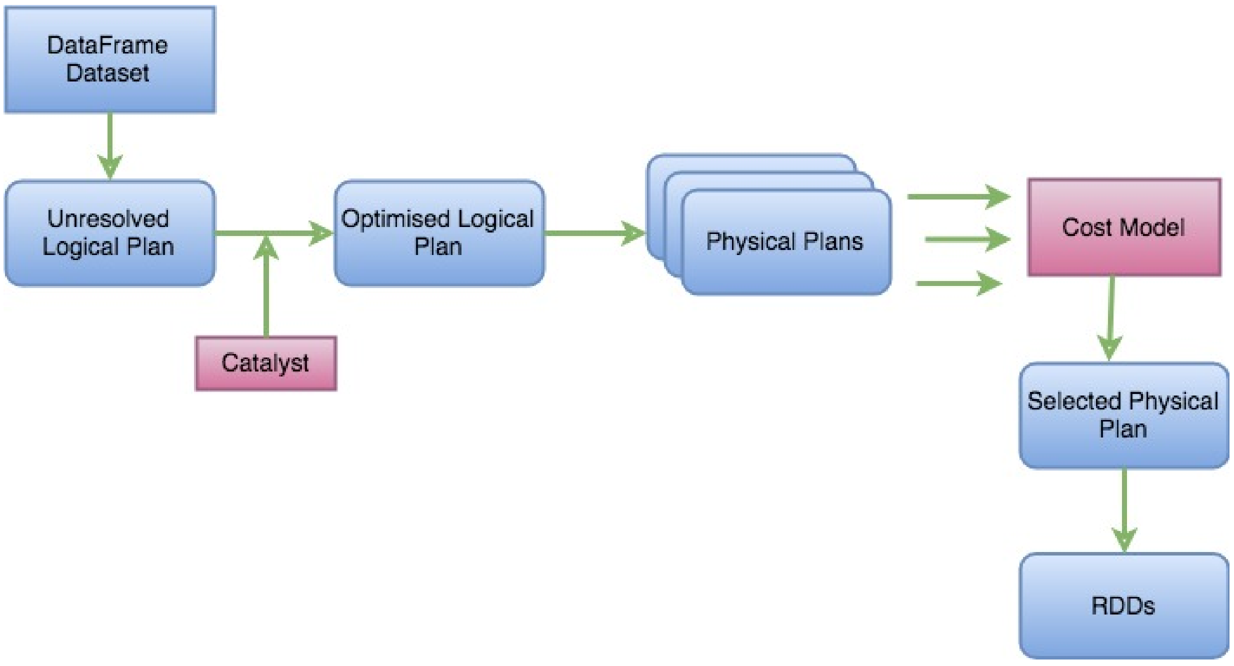

Figure 3.4: DataFrame to RDD optimization

lowing higher-level abstraction and provides a domain specific language API to manipulate distributed data. Datasets is strongly typed and is a map to a relational schema. It represents structured queries with encoders. It is an extension to Dataframe providing both type safety and object-oriented programming interface. It can be constructed from JVM objects and then manipulated using functional transformations.

Dataframes and Datasets are a result of the introduction of an optimizer API known as Catalyst API over the RDD. This API primarily leverages functional programming constructs of Scala such as pattern matching. It offers a general framework for transforming trees, which we use to perform analysis, optimization, planning, and runtime code generation. Figure 3.4 shows the transformation and optimization done within to transform the dataframe and datset to RDD for a more efficient usage of Spark operations. Spark uses catalyst transformation framework in four phases :

- Analyzing a logical plan to resolve references
- Logical plan optimization
- Physical planning
- Code generation to compile the parts of the query to Java bytecode.

3.7.3 Spark Components

Spark Core is the base engine for large-scale parallel and distributed data processing. The core is the distributed execution engine and the Java, Scala, and Python APIs offer a platform for distributed Extract Transformation Load (ETL) application development. Further, additional libraries which are built atop the core allow diverse workloads for streaming, SQL, and machine learning. It provides in-Memory computing and referencing datasets in external storage systems. The major tasks are (1) memory management and fault recovery (2) scheduling, distributing and monitoring jobs on a cluster (3) interacting with storage systems.

Spark Streaming is the component that is used to process real-time streaming data. It enables high-throughput and fault-tolerant stream processing of live data streams. The fundamental stream unit is DStream which is basically a series of RDDs to process the real-time data. It ingests data in mini-batches and performs RDD transformations on those mini-batches of data.

GraphX [36] is the Spark API for graphs and parallel graph computation by extending the Spark RDD with a Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge. This property graph is a directed multigraph which can have multiple edges in parallel. Every edge and vertex have user defined properties associated with it and the parallel edges allow multiple relationships between the same vertices. In order to aid the graph computation, GraphX exposes a set of fundamental operators like subgraph, joinVertices etc., as well as an optimized variant of the Pregel API along with a growing collection of graph algorithms to simplify graph analysis jobs.

Spark SQL integrates relational processing with Spark's functional programming API while supporting querying of data either via SQL or via the Hive Query Language. It integrates relational processing with Spark's functional programming and provides support for various data sources, thus making it possible to weave SQL queries with code transformations. The major contribution of Spark SQL are dataframes and datasets which are dicussed in the previous section.

Spark MLlib [37] is a distributed machine learning library, consisting of fast and scalable implementations of standard learning algorithms for common learning settings including classification, regression, collaborative filtering, clustering, and dimensionality reduction. It also provides a variety of underlying statistics, linear algebra, and optimization primitives. It includes many optimizations to support efficient distributed learning and prediction. Many algorithms benefit from efficient communication primitives; in particu-

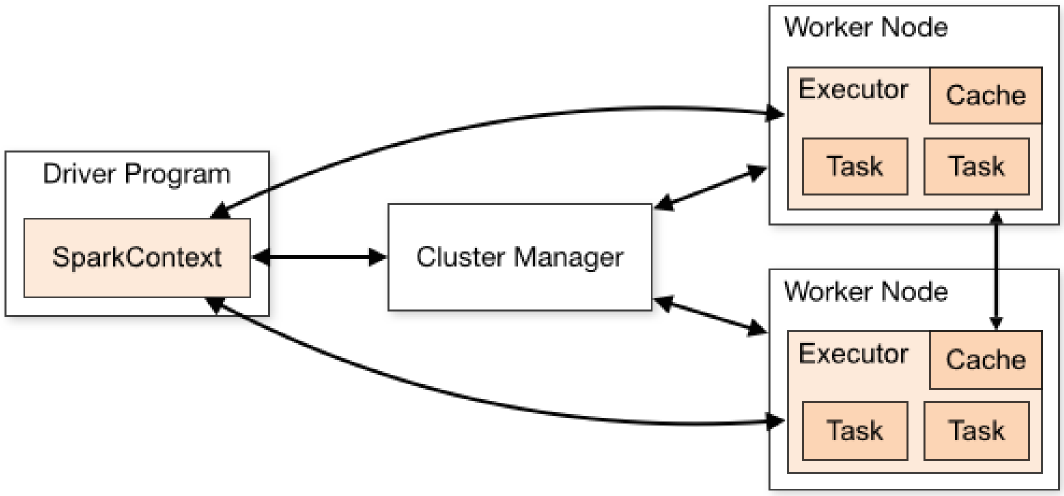

Figure 3.5: Spark Cluster
Source: https://spark.apache.org/docs/latest/cluster-overview.html

lar, tree-structured aggregation prevents the driver from being a bottleneck, and Spark broadcast quickly distributing large models to workers making this library extremely fast and efficient. Being closely integrated with Spark, constant performance improvements in Spark core and these high-level libraries lead to corresponding improvements in MLlib.

3.7.4 Spark Cluster

The figure 3.5 shows an overview of a Spark cluster. Spark works in a master slave architecture and hence has masters and number of workers with configured amount of memory and CPU cores. Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in the driver program. SparkContext can connect to several types of cluster managers that are responsible for allocating resources across the cluster. Once the master is connected to the workers , Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for the application. Finally, it sends the application code (in our case jar files) from SparkContext to the executors.

Each Spark application has its own executor processes, having a lifespan equal to duration of the whole application and running tasks in multiple threads on the available node. As a result, the applications are isolated from each other on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different

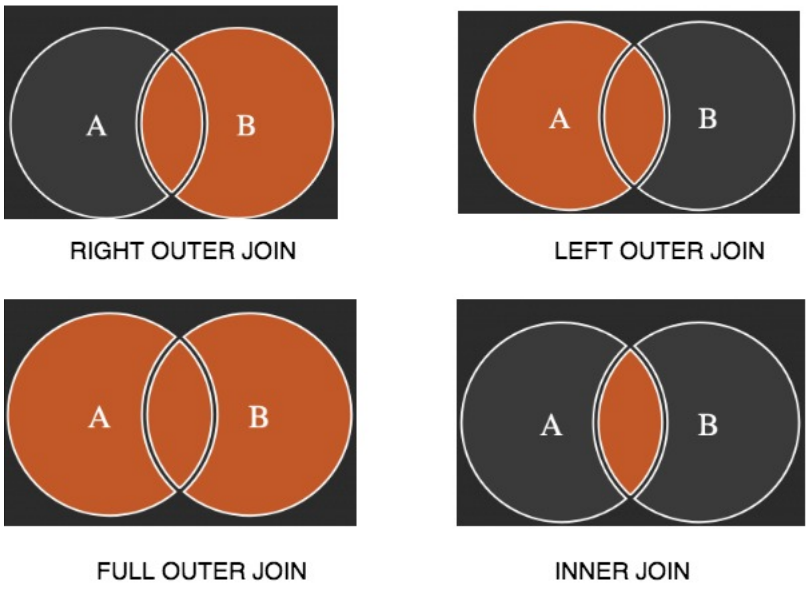

Figure 3.6: Different Types of Joins

JVMs). *However, it also results in data not being shared across different Spark applications without an external storage system.*

3.7.5 Spark SQL Joins

An SQL join clause combines columns from one or more tables in a relational database. It creates a set that can be saved as a table or used as it is. A join is a means for combining columns from one or more tables by using values common to each. Join which uses the same table is a self-join. If an operation uses equality operator, it is equi-join, otherwise, it is non-equi-join. The figure 3.6 shows all the five joins between two dataframes A and B. One join which is difficult to express in Venn diagrams is the Cross Join that produces a result set of the size of the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with it. This kind of result is called as Cartesian Product.

3.7.6 Spark Joins

Joining data is an important part of our EWS pipeline and both Spark Core and SQL support the same fundamental types of joins. While joins are very common and powerful, they warrant special performance consideration as they may require large network transfers or even create datasets beyond our capability to handle.

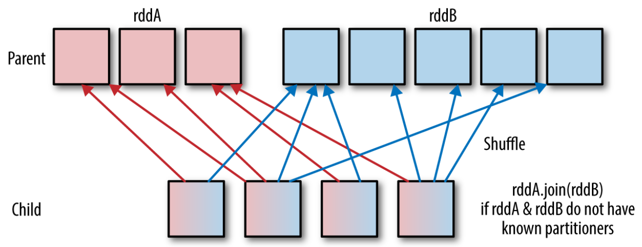

Figure 3.7: Shuffle Join for RDD
Source: High Performance Spark by Rachel Warren, Holden Karau

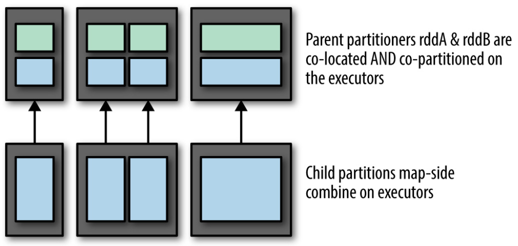
Parent partitioners rddA & rddB are
co-located AND co-partitioned on
the executors
Child partitions map-side
combine on executors

Figure 3.8: Colocated join
Source: High Performance Spark by Rachel Warren, Holden Karau

Inside Spark, a dataframe comprises a fixed number of partitions, each of which comprises a number of records. For the dataframe returned by narrow transformations, the records required to compute the records in a single partition reside in a single partition in the parent RDD/DF. Each object is only dependent on a single object in the parent. Operations like coalesce and joins can result in a task processing multiple input partitions, but the transformation is still considered narrow because the input records used to compute any single output record can still only reside in a limited subset of the partitions. The primary goal when choosing an arrangement of operators is to reduce the number of shuffles and the amount of data shuffled. This is because shuffles are fairly expensive operations; all shuffle data must be written to disk and then transferred over the network.

Joins in general are expensive since they require that corresponding keys from each RDDs are located at the same partition so that they can be combined locally. If the RDDs do not have known partitioners, they will need to be shuffled so that both RDDs share a partitioner, and data with the same keys lives in the same partitions as shown in the figure 3.7.

If RDDs have the same partitioner, the data is colocated, in order to avoid network transfer (Figure 3.8). Regardless of whether the partitioners are the same, if one (or both) of the RDDs have a known partitioner only a

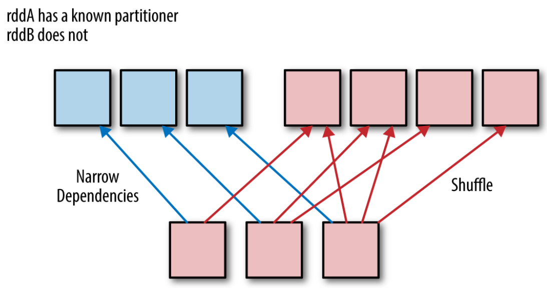

Figure 3.9: Both known partitioner join
Source: High Performance Spark by Rachel Warren, Holden Karau

narrow dependency is created as shown in figure 3.9. As with most key/value operations, the cost of the join increases with the number of keys and the distance the records have to travel in order to get to their correct partition. The joins are effective but very costly because they will cross information between executors, increasing absurdly network transfer and disk usage rate. In the case for very large data sets , the Resource Manager (Yarn) will shutdown the executor with extreme workload, causing a fetch exception when creating new facts.

3.8 SANSA

`SANSA` [38] is an open-source structured data processing engine for performing distributed computation over large-scale RDF datasets. It provides data distribution, scalability, and fault tolerance for manipulating large RDF datasets, and facilitates analytics on the data at scale by making use of cluster-based big data processing engines. It is a wrapper over Apache Spark and Flink frameworks. `SANSA` comes with: (i) specialised serialisation mechanisms and partitioning schemata for RDF, using vertical partitioning strategies, (ii) a scalable query engine for large RDF datasets and different distributed representation formats for RDF, (iii) an adaptive reasoning engine which derives an efficient execution and evaluation plan from a given set of inference rules, (iv) several distributed structured machine learning algorithms that can be applied on large-scale RDF data, and (v) a framework with a unified API that aims to combine distributed in-memory computation technology with semantic technologies. `SANSA` has following layers[39] to achieve its goals ,

- **Knowledge Distribution and Representation Layer**: This is the lowest layer on top of Apache Spark. It provides APIs to load/s-

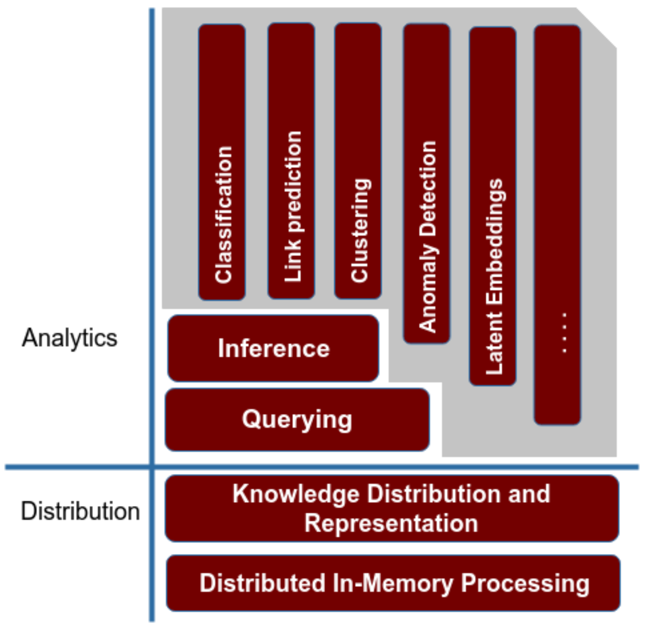

Figure 3.10: SANSA Stack Architecture
Source: http://sansa-stack.net/

tore native RDF or OWL data from HDFS or a local drive into the framework-specific data structures, and provides the functionality to perform simple and distributed manipulations on the data.

- **Query Layer**: SANSA provides APIs for performing SPARQL queries directly in Spark and Flink programs. It also features a W3C standard compliant HTTP SPARQL endpoint server component for enabling externally querying the data that has been loaded using its APIs. These queries are eventually transformed into lower-level Spark programs executed on the Distribution & Representation Layer. It implements flexible triple-based partitioning strategies on top of RDF (such as predicate tables with sub-partitioning by datatypes), which will be complemented with sub-graph based partitioning strategies.

- **Inference Layer**: SANSA supports efficient algorithms for the well-known reasoning profiles RDFS (with different subsets) and OWL, future releases will contain others like OWL-EL, OWL-RL and OWL-LD. In addition, SANSA contains a preliminary version of an adaptive rule engine that can derive an efficient execution plan from a given set of inference rules by generating, analysing and transforming a rule-dependency graph.

- **Machine Learning Layer**: The machine learning algorithms in `SANSA` exploit the graph structure and semantics of the background knowledge specified using the RDF and OWL standards. The machine learning layer contains distributed implementations of link prediction algorithms based on two knowledge graph embedding models, namely Bilinear-Diag and TransE, and scalable algorithms for RDF data clustering and association rule mining. It also consists of implementation of rule mining algorithm like AMIE+ .

We hope this work to be a part of `SANSA` machine learning layer. Figure 3.10 shows an overview of `SANSA`'s architecture.

3.9 Decision Tree Classifier

Decision tree algorithm [40] is a data mining induction technique that recursively partitions a set of records using either depth-first or breadth-first approach until all the data items belong to a one or other class.DT classification algorithms can be implemented in a serial or parallel fashion based memory space available on the computational, on the volume of data, and scalability of the algorithm. Decision trees are especially attractive for a data mining environment for the following reasons.

1. Due to their intuitive representation, they are easy to assimilate by human
2. They can be constructed relatively fast compared to other methods.
3. The accuracy of decision tree classifiers is comparable or superior to other models

The input is a database of training records [41]such that each record has several attributes. Attributes whose underlying domain is totally ordered are called *ordered attributes*, whereas attributes whose underlying domain is not ordered are called *categorical attributes*. There is one distinguished attribute, called *class label*, which is a categorical attribute with a very small domain. The remaining attributes are called *predictor attributes* and are either ordered or categorical in nature. The goal of classification is to build a concise model of the distribution of the class label in terms of the predictor attribute. More precisely, a decision tree algorithm's objectives are:

- Classify the maximum number of training samples correctly.

- Generate a decision logic beyond the training sample so that unseen samples could be classified with highest possible accuracy.
- The decision logic should be easy to update with the availability of more training sample.
- Tree has a simple structure.

The training data set is used in building the classification model, while the test data record is used in validating the model. The model is then used to classify and predict new set of data records that is different from both the training and test data sets.

The work [42, 43] provide a detailed survey of various decision tree classification algorithm and its variants.

3.10 Link Prediction

Link prediction is a learning task in charge of finding new relations within a knowledge graph. Generally, the link prediction problem can be classified into two categories: (1) predict the likelihood of a future link or missing link between two nodes, knowing that there is no link between them in the current graph, and (2) predict whether there will be a future interaction between two nodes that have an association in the current graph. Formally,

Definition 3.10.1. Link Prediction Given a KG, G, as defined in 3.2.1, the task of link prediction is to identify for each predicate p and for any resource s ,another resource o such that $p(s, o)$ is in G and $u, v \in V$; or alternatively, to identify for each predicate P and each entity v, an entity u such that $P(u, v)$ is in the KG.

Since we don't know which edges are the missing or future edges in order to test the algorithm's accuracy, the observed edges, E, is randomly divided into two parts: the training set, E^T, is treated as known information, while the validation subset,E^P, is used for testing and no information in this set is allowed to be used for prediction. This implies, $E^T \cup E^P = E$ and $E^T \cap E^P = \phi$. Since this method has a statistical bias, some other methods like *K-fold cross-validation* are used. In this evaluation methodology, the observed links are randomly partitioned into K subsets. Each time one subset is selected as probe set, the remaining $K - 1$ constitute the training set. The cross-validation process is then repeated K times, with each of the K subsets used exactly once as the validation set. In this work, we evaluate the predictive power of mined rules for link prediction for few KGs.

Chapter 4

Learning Exception-enriched Rules

In this chapter, we discuss in detail the algorithm [5] and implementation of the mining of exception enriched rules on Spark. We recapture certain definitions required for a better understanding of the algorithm. For further details, please refer to the original paper.

4.1 Problem

Given an incomplete graph G^i, as defined in def 3.2.1, containing a set of facts of the form $\langle s, p, o \rangle$. and a set of Horn rules R_H, the goal of the algorithm is to mine negated atoms (exceptions) to the rules in R_H and output a revised rule-set R_N such that the difference between the number of correct facts between G^i and the complete graph G^c (containing all correct facts) is reduced.

The original algorithm to mine EWS [5] is implemented in Java using SPMF pattern mining library [44] implemented on a single multi-core system. Since all the data is available on the standalone system the nuances of data distribution has not been taken into consideration. Further the size of the dataset and power of computations are bound by a single system. To overcome the memory and computational limitations of a single system, we aim to achieve the goal of mining exceptions in a distributed cluster based environment using Spark. However, the original algorithm cannot be implemented as it is in a Spark because of the aforementioned limitations.

Moreover, as mentioned in the section 3.7, Spark works in a master slave architecture. The primary unit of Spark i.e., `RDD` and Dataframe are driver-side (master) abstractions of distributed collections. They cannot be used, created, or referenced in any executor-side (slave) transformation. They can only be used within the context of an active SparkSession outside which, the DataFrame (in our case) cannot reference to its partitions on the active executors. The SparkSession can be thought of as an active connection of the master to the cluster of executors. When an algorithm tries to access a DataFrame inside another transformation, that DataFrame would have to be serialized on the driver side, sent to the executor(s), and then deserialized there. But this deserialized instance (in a separate JVM) necessarily loses it's corresponding original SparkSession (from the driver to the executor) and replaces it with the new one (from this new executor). Spark does provide solutions for global variables by providing in-memory storage of these variables and finally broadcast throughout the cluster. However, the size of the data required to be broadcast is large enough to be stored into in-memory. Hence, the algorithm to mine EWS in a distributed environment requires a reiteration right from the definition stage.

4.2 Approach

We assume an input of a incomplete knowledge graph G in accordance with def 3.2.1 like in the example shown in figure 4.1. Firstly, we mine Horn rules using a Spark based implementation of FP-Growth [45]. This mining algorithm could have very well been replaced by any other popular rule mining approach like AMIE+ [4] or HornConcerto [27]. Since the original work mentioned the limitations of AMIE mined rules, and even though the implementation works independent of Horn Rules mining technique we chose FP-Growth primarily as it gave a wide variety of rules to be refined and its implementation in Spark has been optimized to a great level as compared to the alternatives. For our example, we assume the following rule as an output of FP-Growth,

$$worksAtUniBonn(X) \wedge teachesClass(X) \implies profAtUniBonn(X)$$

Once we obtain the Horn rules, for each rule we calculated a positive and negative instance set which are defined as follows:

Definition 4.2.1. R-Positive Instance Set Let F be the set of facts on a knowledge graph G and $r : b_1(X) \wedge b_2(X) \wedge .....b_n(X) \implies h(X)$ be a horn rule mined on G, then the positive instance set is defined as a set of facts which are true (exists in) for the antecedant and consequent of the rule r. Formally,

$$P(r, F) = \{x | b_1(x), b_2(x), ...b_n(x), h(x) \in F\}$$

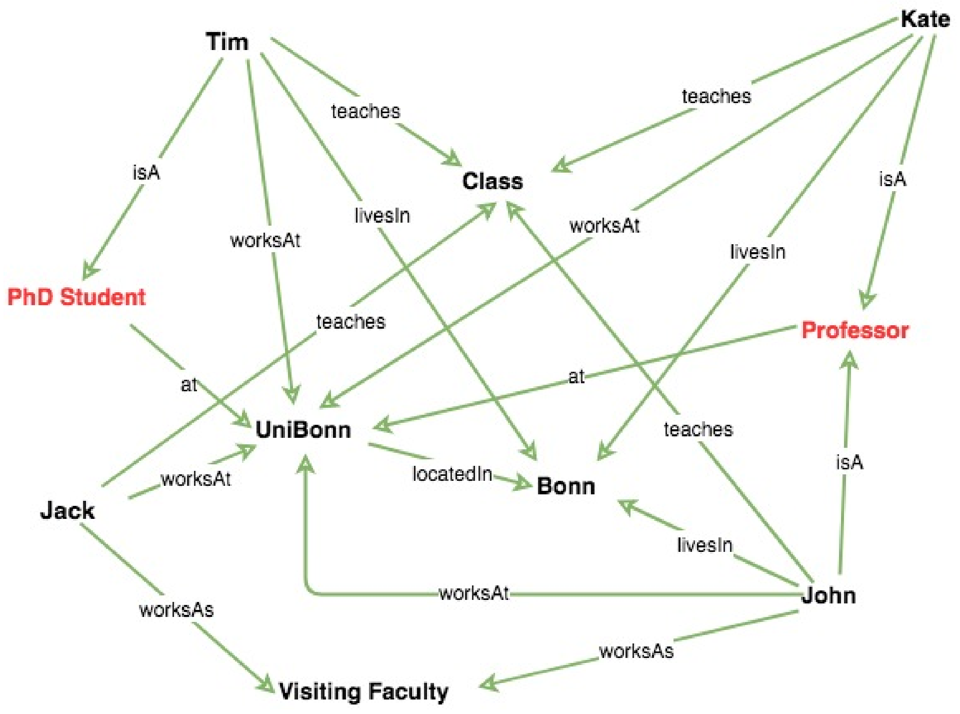

Figure 4.1: Example `RDF` KB

where $x \in F$

Definition 4.2.2. R-Negative Instance Set Let F be the set of facts on a knowledge graph G and $r : b_1(X) \wedge b_2(X) \wedge .....b_n(X) \implies h(X)$ be a horn rule mined on G, then the negative instance set is defined as a set of facts which are true (exists in) for the antecedant but is false (does not exist) for consequent of the rule r. Formally,

$$N(r, F) = \{x | b_1(x), b_2(x), ...b_n(x), \in F and \\ h(x) \notin F\}$$

where $x \in F$

For the example rule R_1, the R-Positive Instance Set $(P(R_1, F))$ is $\{John, Kate\}$ while the R-negative instance set $(N(R_1, F))$ is $\{Tim, Jack\}$

In case of ideal graph G^i, the R-positive and R-negative instance sets would exactly correspond to instances for which the rule r holds (resp. does not hold) in the real world. But since we are working with an non-ideal/ incomplete graph G some members of R-negative instance set might exist because of data incompleteness. This leads to the following definition of Exception Witness Set,

Definition 4.2.3. Exception Witness Set Let F be the set of facts on a knowledge graph G and $r : b_1(X) \wedge b_2(X) \wedge .....b_n(X) \implies h(X)$ be a horn rule mined on G, then r-exception witness set $EWS(r, F) = \{e_1, e_2, e_3..e_l\}$ is a maximal set of "exceptional" predicates such that,

- $\exists x \in N(r, F) \mid e(x) \in F$ and,
- $\forall x \in P(r, F) \mid e_1(x), e_2(x), e_3(x)....e_l \notin F$

From our examples, the exception predicates were $\{isAPhDStudent$, $isAVisitingFaculty\}$. Based on the definition, we need a predicate which has no fact belonging to R-Positive Instance Set. In this case, $isAVisitingFaculty$ has one fact (i.e., John) which belongs to $P(R_1, F)$. Hence the $EWS(R_1, F) = \{isAPhDStudent\}$.

Once the EWS are computed for all horn rules, we append them to the corresponding rules as negated body elements and check the new rules based on confidence.

Thus, the refined rule becomes,

$$worksAtUniBonn(X) \wedge teachesClass(X) \wedge \neg isAPhDStudent(X) \\ \implies profAtUniBonn(X)$$

4.3 Architecture and Implementation

Figure 4.2 shows the pipeline of the Exception Enriched Rule Miner (EERM). Firstly, we use SANSA [39, 46] to read the dataset and translate it into a Spark dataframe, (described in Chapter 3). SANSA uses the RDF data model for representing graphs consisting of triples with subject, predicate and object. RDF datasets may contains multiple RDF graphs and record information about each graph, allowing any of the upper layers of SANSA (Querying and ML) to make queries that involve information from more than one graph. Instead of directly dealing with RDF datasets, the target RDF datasets need to be converted into an RDD/DataSets of triples. The main dataset is based on an RDD/DataSets data structure, which is a basic building block of the Spark framework. RDDs/DataSets are in-memory collections of records that can be operated on in parallel on large clusters. This dataset is available "lazily" to the other modules of the EERM. Once the dataset is loaded into dataframe, we concatenate the predicates and objects from the triples to create unary predicates.

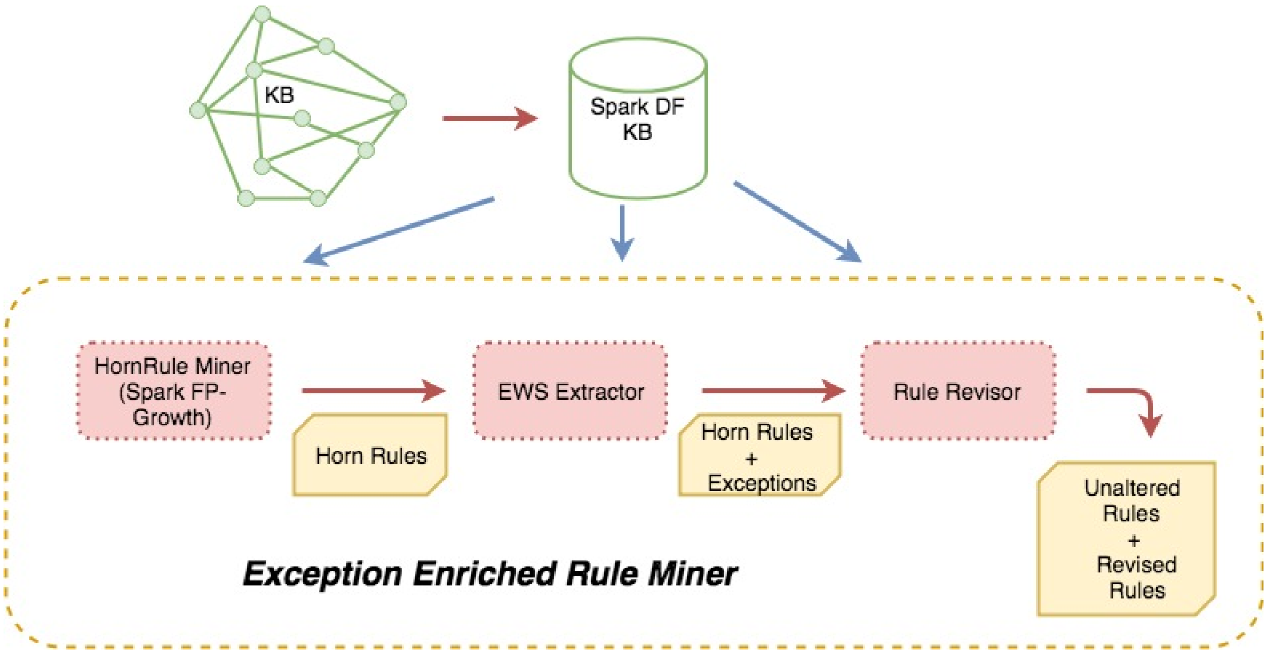
KB
Spark DF KB
HornRule Miner (Spark FP-Growth)
Horn Rules
EWS Extractor
Horn Rules + Exceptions
Rule Revisor
Unaltered Rules + Revised Rules
Exception Enriched Rule Miner

Figure 4.2: Exception Enriched Rule Miner Architecture

4.3.1 Mining Horn Rules

The FP-Growth in Spark is variant of traditional FP-Growth Algorithm , a parallel version of FP-growth called PFP [47]. Frequent itemset mining (FIM) is a useful tool for discovering frequently co-occurrent items.

FP-Growth [37] works in a divide and conquer manner. It requires two scans on the database. FP-Growth first computes a list of frequent items sorted by frequency in descending order during its first database scan. In its second scan, the database is compressed into a FP-tree. Then FP-Growth starts to mine the FP-tree for each item whose support is larger than ξ by recursively building its conditional FP-tree. The algorithm performs mining recursively on FP-tree. The problem of finding frequent itemsets is converted to searching and constructing trees recursively. The dataset containing the unary predicates in converted into a dataset of transactions as described as follows.

Let $I = \{a_1, a_2, \ldots, a_m\}$ be a set of items (in our case predicates), and a transaction database DB is a set of subsets of I, denoted by $D = \{T_1, T_2, ..., T_n\}$, where each $T_i \subseteq I$ $(1 \leq i \leq n)$ is a transaction. The support of a pattern $A \subseteq I$, denoted by `supp(A)`, is the number of transactions containing A in D. `A` is a frequent pattern if and only `supp(A)`$\geq \xi$, where ξ is a predefined minimum support threshold. Given D and ξ, the goal is to find the complete set of frequent patterns.

This miner module creates several Spark jobs in order to execute FP-Growth algorithm on the KG. Given a transaction database D, Parallel FP-Growth uses three MapReduce phases to parallelize FP-Growth. The transaction database is processed in following steps:

1. Divide D into successive parts and store the parts on P different node. Such division and distribution of data is known as sharding, and each part is called a shard.

2. Implement a MapReduce pass to count the support values of all items that appear in D. Each mapper inputs one shard of D. This step implicitly discovers the items' vocabulary I, which is usually unknown for a huge D. The result is stored in *Frequency List*.

3. Divide all the $|I|$ items on *Frequency List* into groups. The list of groups is created (*Group List*), where each group is given a unique group id . As both the lists are small and the time complexity is $O(|I|)$, this step can complete on a single node in few seconds.

4. This step takes one MapReduce pass, where the map stage and reduce stage perform different tasks:

    - **Mapper – Generating group-dependent transactions** Each mapper instance takes a shard of DB generated in the previous step. Before it processes transactions in the shard one by one, it reads the *Group List*. It outputs one or more key-value pairs, where each key is a group-id and its corresponding value is a generated group-dependent transaction.
    - **Reducer – FP-Growth on group-dependent shards** Once all mapper instances have finished their work, for each group-id, the MapReduce infrastructure automatically groups all corresponding group-dependent transactions into a shard of group-dependent transactions. Each reducer instance is assigned to process one or more group-dependent shard one by one. For each shard, the reducer instance builds a local FP-tree and growth its conditional FP-trees recursively. During the recursive process, it may output discovered patterns.

5. Aggregating the results generated in Step 4 as the final result.

To summarize, FP-growth calculates item frequencies and identify frequent items following which it uses a suffix tree (FP-tree) structure to encode transactions without generating candidate sets explicitly, which are usually expensive to generate. After the second step, the frequent itemsets can be extracted from the FP-tree. The Parallel FP-Growth distributes the work of growing FP-trees based on the suffixes of transactions. This module finally outputs a set of horn rules. These horn rules are passed on the EWS Extractor module, which is primarily responsible for extraction the exceptions as defined Definition 4.2.3.

4.3.2 EWS Miner

Initial Approach

As per the original algorithm, a variant of induction learning [48] (frequent pattern mining) was used to mine the frequent patterns. Our initial approach was very similar and is described as follows:

1. Firstly for each rules r, we calculate $N(r_h, F)$ and $P(r_h, F)$ where F is the facts dataframe as defined in previous section and stored them.

2. Following the previous step, the facts for predicate $nota(c)$ are added to a new set $E+$ for all $c \in N(r_h, F)$. In the same step the facts not a(c) for $c \in P(r_h, F)$ are stored in $E-$.

3. In a variant of a classical inductive learning procedure, `Learn`($E+$,$E-$,F), is employed to induce a set of exception witness set candidates R_e in the form of Horn rules with unary atoms,

4. Finally, the bodies of rules in R_e not containing predicates from positive body elements of r are put in EWS, which is output.

However, the procedure didn't take into account the distribution of data across the nodes. Since there are no out of the shelf implementation of any similar procedure, the implementation required use of User Defined Functions (UDFs). UDFs are great when built-in SQL functions are not sufficient. There are several limitations to using a UDF.

- The number of parameters that can be passed in an UDF on are limited.
- UDFs must be self sufficient to a degree; most of UDF logic must be done within the UDF or another UDF.
- UDFs can return only one result set

- UDFs cannot execute dynamic SQL or temporary table and as a result it cannot be used to change, extract and transform other dataframe other than the dataframe which calls it.

However, the approach requires an iterative process over the entire rule dataframe and access to the KB dataframe several times within each iteration of the loop over the rules. Even though the algorithm is quite efficiently created, it cannot be implemented in the Spark. This process requires passing of the facts dataframe, horn rules dataframe obtained from the previous steps. These dataframes increase larger in size with increase in number of triples. Moreover, the initialization and calling of pattern mining algorithm from Spark shelf for each rule. Both these factors violates the limitation of UDF being self sufficient

Moreover, as mentioned in the section 3.7, Spark works in a master slave architecture. A Spark application consists of a single driver process and a set of executor processes scattered across nodes on the cluster. The driver is the process that is in charge of the high-level control flow of work that needs to be done. The executor processes are responsible for executing this work, in the form of tasks, as well as for storing any data that the user chooses to cache. Both the driver and the executors typically stick around for the entire time the application is running, although dynamic resource allocation changes that for the latter. A single executor has a number of slots for running tasks, and will run many concurrently throughout its lifetime. Deploying these processes on the cluster is up to the cluster manager in use (YARN, Mesos, or Spark Standalone), but the driver and executor themselves exist in every Spark application. Inside the UDF, which are executor-side (slave) transformation, the dataframes cannot be used, created, or referenced. When the algorithm tries to access the facts dataframe inside the UDF, that DataFrame would have to be serialized on the driver side, sent to the executor(s), and then deserialized there. But this deserialized instance (in a separate JVM) necessarily loses it's corresponding original SparkSession (from the driver to the executor) and replaces it with the new one (from this new executor). Moreover these dataframes are too large to be used as broadcast variable throughout the cluster. Hence, the approach failed miserably and ran out of memory on the driver side when used applied in the cluster mode. This approach did work in a standalone mode of Spark as all the data exists between only on the same worker and driver.

Final Approach

In order to overcome the limitations of our initial approach we implemented a more intuitive approach for mining of Exception Witness Set. The ease of dataframe based set operations in Spark SQL [35] helps us in calculation of EWS via joins. The use of joins gives us the liberty to process all the rules in parallel instead of iteratively going through each rule. Figure 4.3 shows the transformation diagram of dataframes. The transformation to obtain EWS are described as follows:

1. The `Horn RuleDF` dataframe and two transformed map dataframe of facts from KB were taken as input. These maps are:
   - `Operator-Subject MapDF` It is a map from unary Operator to facts .
   - `Subject-Operator MapDF` It is a map from facts to unary operator.
2. We join the `Horn RuleDF` with `Operator-Subject MapDF` based on the operators that occur in body to obtain all the facts that occur with the body of each rule i.e., the $N(r_h, F)$ set as explained in the Approach section.
3. We join the `Horn RuleDF` with `Operator-Subject MapDF` based on the operators that occur in head to obtain all the facts that occur with the head of each rule i.e., the $P(r_h, F)$ set as explained in the Approach section.
4. We join the dataframes created in step 2 and 3 to get all the rules with body and head facts in a final dataframe, `RulesFactsDF`.
5. We calculate the difference in the column of body and head facts to get facts that exclusively occur in body.
6. The facts obtained from previous step are joined to `Subject-Operator MapDF` to obtain the operators with a join with `Operator-Subject MapDF` which satisfy the definition of EWS as mentioned in Def. 4.2.3.
7. These EWS are filtered based on support count.

To summarize, we follow the intuitive of EWS from the definitions of $P(r_h, F)$ and $N(r_h, F)$, we calculate the difference in facts between $N(r_h, F)$ and $P(r_h, F)$ and return the predicates for which these facts exists. These predicates are further filtered based on confidence scores and serve as the exception for the rule r_h.

The exceptions obtained from the second step along with original rules are passed on to the Rule Revision module. The Rule Revision module checks for the confidence of the these rules and finally return the rules having confidence more than a certain threshold. These rules are used to generate new facts.

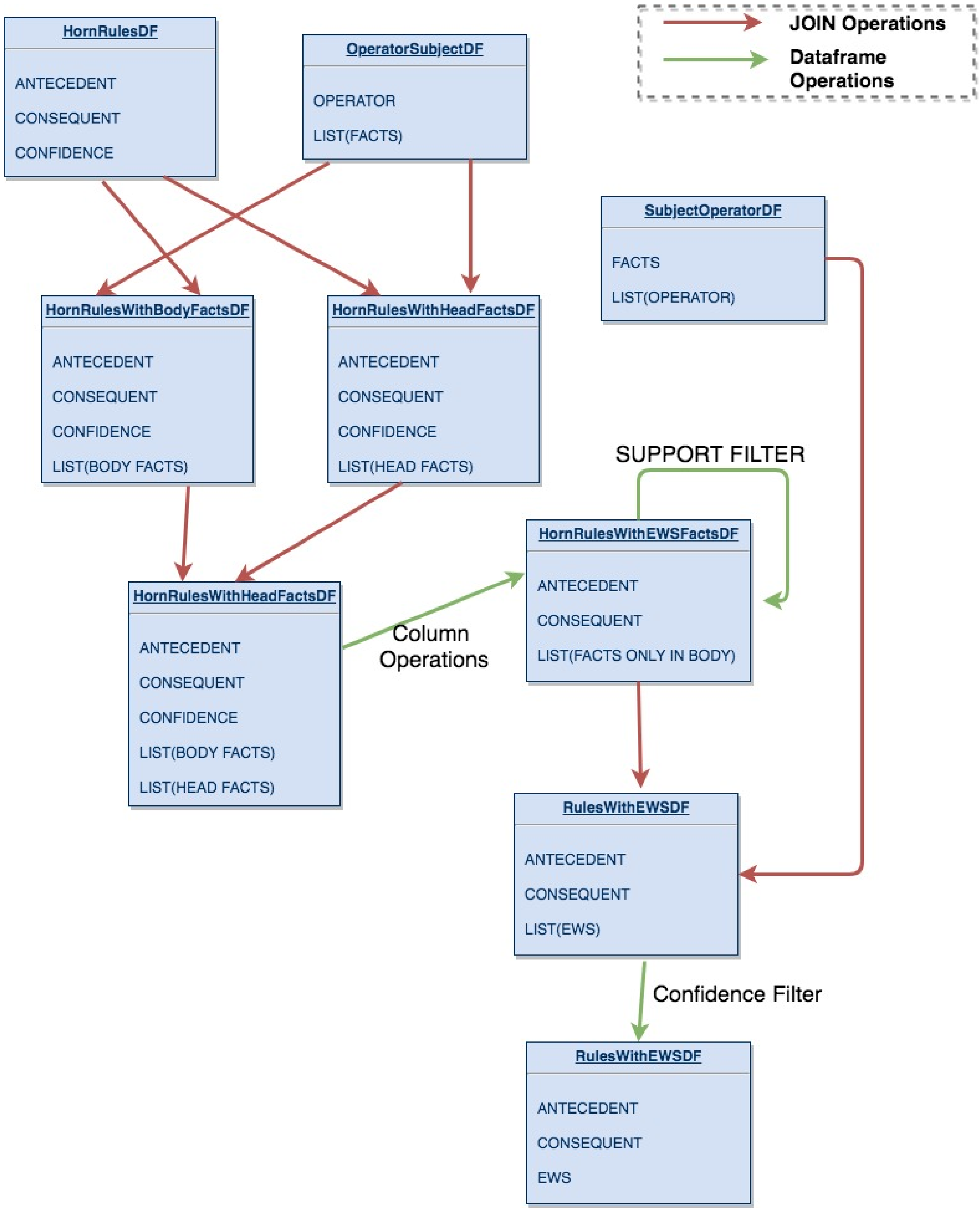

Figure 4.3: EWS Mining Approach

Fact Generator

The resultant dataframe containing the final rules in the form $\langle body, negative, head\rangle$ is used to generate new facts by joining it with the `Operator-Subject MapDF` such that the facts that occur for both the *body* and *negative* operators must also occur for the head. The head operator split into predicate object pair along with the fact as subject serves as the new triple.

Chapter 5

Rule Mining using Decision Tree in Spark

In this chapter, we present a novel approach of using Decision Tree for rule mining. We classify rules as containing only positive body atoms, containing only negative body atoms and containing both positive and negative body atoms. We try to mine all the three types of rule using classical Decision tree [49] approach.

5.1 Problem

Given an incomplete graph G^i, as defined in def 3.2.1, containing a set of facts of the form $\langle s, p, o \rangle$, our goal is to mine a set of rules R_G, not necessarily Horn rules, such that the difference between the number of correct facts G^i and the complete graph G^c (containing all correct facts) is reduced.

The rule set R_G can be formally defined as,

$$\bigwedge_{i=0}^{n} b_i \wedge \bigwedge_{i=0}^{m} \neg e_i \implies h \tag{5.1}$$

where $B = \{b_0, b_1, b_2 ... b_n\}$, are positive body predicates , where $E = \{e_0, e_1, e_2$
$...e_m\}$ are exceptions and combine to form the negated part of the body and $H = \{h\}$ is the head. Here, $b_0 = \perp$ and $e_0 = \perp$ and hence both B and E can be null set.

		Predicates					
		teachesClass	livesInBonn	worksAtUniBonn	isAPhDStudent	isAProfessor	worksAsVistingFaculty
Subjects	John	✓	✓	✓		✓	✓
	Tim	✓	✓	✓	✓		
	Kate	✓	✓	✓		✓	
	Jack	✓		✓			✓

Table 5.1: KB Sparse Matrix

5.2 Approach

Given a knowledge base G, for example figure 4.1, we first translate the predicate and object pair to obtain unary predicates following which we transform the KB in a matrix K with the dimension $|P| \times |S|$, where $|P| = \#unaryPredicate$ i.e., P is the set of unary predicates and $|S| = \#subjects$ with S as a set of subjects. In K, each column is a unary predicate $h \in P$, and row correspond to the subject $s \in S$. The matrix for the example KB, as shown in figure 4.1 is shown in Table 5.1. We iteratively treat each of the column h as the classification property and the remaining columns as training vectors and create decision tree with predicate h as head (label predicate) and the path of the decision tree from the root to the leaves serves as the body of the rule.

A row in the matrix K is formed by the entries $(x, Y) = \{x_1, x_2 \dots x_n, Y\}$, where x_i is a training predicate vector such that $x_i \in 0, 1$ with $i = 1 \dots n$ and Y is a label vector such that $Y \in [0, 1]$. Given this row we try to find a predictive model function $f : Y \approx f(x)$ via recursively partitioning the space such that the samples with the same labels are grouped together.

Let the data at tree node m be represented by J. For each candidate split $\theta \in \{0, 1\}$, partition the data into $J_{left}(\theta)$ subtree and $J_{right}(\theta)$ subtree subsets defined as follows,

$$J_{left}(\theta) = (x, y) | x_m = 1$$

$$J_{right}(\theta) = J \setminus J_{left}$$

The impurity, measure of the homogeneity of the target variable within the subsets at m, is computed using Gini index impurity function such that,

$$H(x_m) = \sum_k p_m(1 - p_m)$$

where p_m is defined as the proportion of a presence of target predicate w.r.t total observation in node m and $k \in \{0, 1\}$. Formally,

$$p_m = \frac{1}{N_m} \sum_{x_i \in \{0,1\}} f(x|y_i = 1)$$

The learned tree T produces a set of paths of $\Pi_T = \{\pi_1, \pi_2 \ldots \pi_n\}$, where each path π can be seen as array of tuples consisting of predicate j and a decision $d_i \in \{\top, \bot\}$. Each path ends with along with a leaf node $L \in \{\top, \bot\}$.

Hence the path is given by,

$$\pi = \{(j_0, d_0), (j_1, d_1), \ldots (j_n, d_n), L\}$$

where j_0 is the root node. Finally, each path π where $L = \top$ is taken as body of the rule of the form $B \wedge E \implies H$, where

- $B = \{j|(j, d) \in \pi \wedge d = \top\}$,
- $E = \{\neg j|(j, d) \in \pi \wedge d = \bot\}$ and
- $H = h$

These rules are used to generate new facts which are evaluated via Link Prediction (discussed in chapter 6).

5.3 Architecture and Implementation

The figure 5.1 shows the architecture of Decision Tree Rule Miner (DTRM). Firstly, we use `SANSA` [39, 46] to read the dataset and translate it into a Spark dataframe, (described in Chapter 3) which is available "lazily" to the other modules of the DTRM. Once the dataset is loaded into dataframe, we concatenate the predicates and objects from the triples to create unary predicates and translate it into a sparse matrix K with each column as a unary

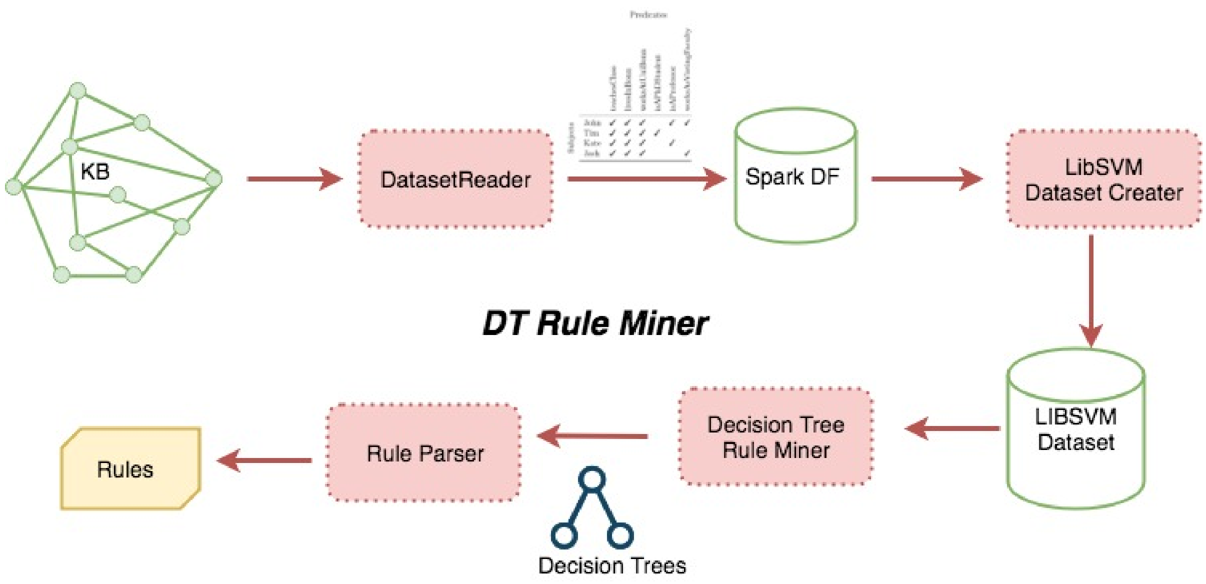

Figure 5.1: Decision Tree Rule Miner Architecture

relation and row for each subject. The units of the matrix are filled with either 0 or 1. A single unit is 1 if subject in the row occurs with the corresponding unary predicate in the column, otherwise it is 0. The dimensions of this matrix ideally should be $\#unaryPredicate \times \#subjects$ as described in the previous section. However, we reduce the number of predicates based on support count for each predicate. Furthermore, we rely on Spark sparse matrix optimization techniques to efficiently handle to storage of large matrices. This matrix is stored in dataframe which is available "lazily" to the next modules of the DTRM.

LIBSVM Dataset Creator

Once we have the matrix loaded into dataframe we transform the dataframes to the matrix into LIBSVM format [50] with LIBSVM Dataset Creator module.

LIBSVM format is primarily useful for representing the sparse matrix. Essentially, it stores only the non-zero data, and any missing data is taken as holding null/zero value. The LIBSVM format looks like

$$\langle label\rangle\langle index1\rangle : \langle value1\rangle\langle index2\rangle : \langle value2\rangle \ldots$$

Each line contains an instance. For classification, $\langle label\rangle$ is an integer indicating the class label. Feature space is a space for the multidimensional data. A feature is simply an indicator to associate or correlate your target value with, so a better prediction can be made. Each feature (vector) should have its own ID (index) and its value. The index marks a feature of the data and its value. It merely serves as a way to distinguish between the features/parameters.

The LIBSVM Dataset creator takes the KB as input. We create a dataframe containing all the operators with their id numbers known as `OperatorToID`. For each predicate, h, in the KB, we treat h as the head of the rule and treat the corresponding column in the matrix as class label (see section 3.9). The remaining predicates are taken as they are and the whole data is translated in the LIBSVM format based on their presence with the head predicate h. These files are then written to the HDFS cluster to be used further. These files are the equivalent to the sparse matrix representation as mentioned in the previous section.

Decision Tree Rule Miner

We launch Spark jobs to run Decision Tree [37] (in Decision Tree Rule Miner module) on these LIBSVM files. For each head predicate h, we create decision trees and extract paths $\Pi = \{\pi_1, \pi_2, \pi_3, \dots\}$ as described in the previous section. Each path π_i with the leaf value $L = \top$ is the body of the rule as described in the previous section. Hence the output of the tree,

$$\pi_i \implies h$$

Since $\Pi = \Phi$ is also possible, tree can return 0, 1, or more rule(s).

Spark uses various optimizations such as data-dependent feature discretization to reduce communication costs, and tree ensembles parallelize learning both within trees and across trees. At the lowest level, Spark core provides a general execution engine with over 80 operators for transforming data, e.g., for data cleaning and featurization. MLlib also leverages the other high-level libraries packaged with Spark. We try to capture the idea of decision tree internals inside Spark.

In a Decision Tree, every internal node means a test on an attribute, every branch means the output of a test, and every leaf node store a class label. Spark adopts the top-down and recursive method to construct a decision tree from the training items and the categories they belongs to. The steps for a creation of decision tree are as follows:

1. Read the LIBSVM file where each line is an item. Each item has some attribute values and a class label;

2. Find the gain ratio from splitting on each attribute `att`;

3. Let `attribute_best` be the attribute with the highest gain ratio;
4. Create a decision node that splits on `attribute_best`;
5. After splitting on `attribute_best`, rest of the attributes are brought into consideration for the child node which is distributed over partitions. For each partition go back to step 2) to get `attribute_best_child` for the child node. `Attribute_best` will be the child of the node formed in 4).
6. The steps 1 to 4 are repeated till the desired accuracy rate is reached.

The Decision Tree Rule Miner module return a dataframe with `RulesWithIds`. We use the `OperatorToId` dataframe created by LIBSVM Dataset Creator module to get the rules in the form of operators. It involves joining the `OperatorToId` and `RulesWithIds` followed by a groupBy operation.

Following our previous example, for the head $profAtUniBonn(X)$ a sample tree is shown in figure 5.2. Hence we get the rule,

$$\begin{aligned} &livesinBonn(X) \wedge worksAtUniBonn(X) \wedge teachesClass(X) \wedge \\ &\qquad \neg isAPhDStudent(X) \wedge \neg worksAsVistingFaculty(X) \\ &\qquad\qquad\qquad\qquad \implies profAtUniBonn(X) \end{aligned}$$

In the example, there is only one path that results in a 1 confidence leaf node but in reality there can be zero, one or more leaves with confidence 1. As evident from the example, the *"No"* branches are treated as negated body atoms while the *"yes"* branches form the positive body atoms. Once the rule set are obtained we calculate the confidence score to and filter the rules not having a minimum support count and confidence.

Fact Generator

The resultant dataframe containing the final rules in the form ⟨*body*, *negative*, *head*⟩ is used to generate new facts by joining it with the `Operator-Subject MapDF` such that the facts that occur for both the *body* and *negative* (in case where *negative* column is not null) operators must also occur for the head. The head operator split into predicate object pair along with the fact as subject serves as the new triple.

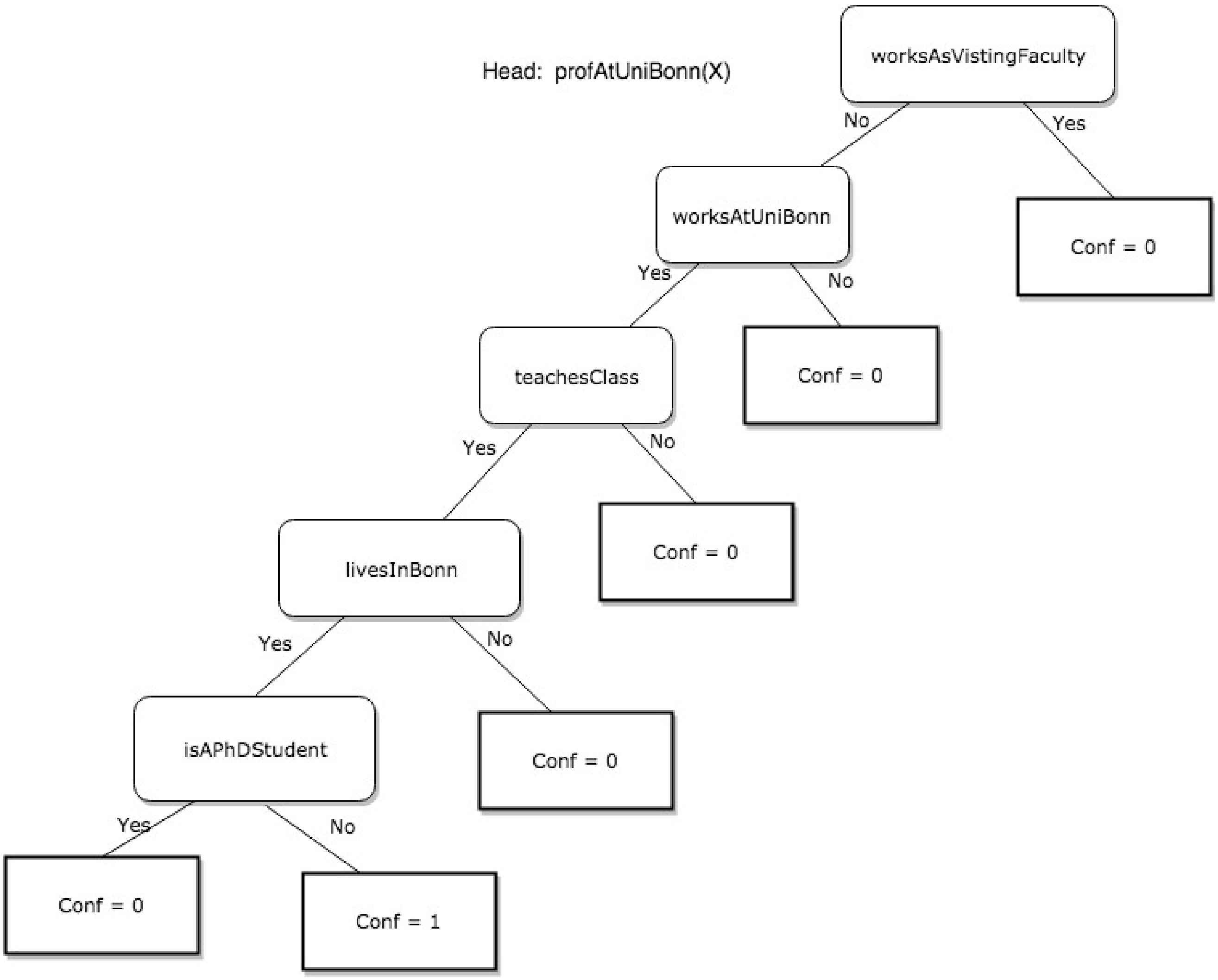

Figure 5.2: Decision Tree Result for Figure 4.1

Chapter 6

Evaluation

This chapter covers the evaluation of both the Exception Enriched Rule Mining and Decision Tree Rule mining algorithm. We evaluated the rule mining algorithms on the basis of following metrics (1) execution runtime, (2) number of rules discovered, (3) quality of rules based on human evaluation, and (4) quality of rules for link prediction (for EERM).

6.1 Exception Enriched Rule Mining Results

6.1.1 Experimental Setup

Table 6.1: Datasets used in the evaluation

Dataset	# triples
WordNet (WN18)	146K
Freebase	533K
IMDb	2 Million
YAGO	10 Million

We perform the analysis of our approaches based on the datasets used are described in Table 6.1. WordNet and Freebase are part of a benchmark for link prediction [51]. Both datasets are divided into two parts, i.e. training, and test set. The sizes shown in the table are the union of the training and validation sets, while the test sets feature 5K and 60K triples respectively.

We also evaluate our system on a slice of almost 10M facts from YAGO3, a general purpose KG, and an RDF version of IMDB data with 2M facts, a well known domain-specific KG of movies and artists. The choice of the KBs allows us to evaluate our method's performance on both general-purpose and domain-specific KGs.

We test the system on a three node Spark cluster where we initiated 35 executors in the cluster with 5 cores and 19 GB per executors. Some preparations are made before executing the driver program. YARN and Spark environment needs to be deployed, and the training data should have been put on HDFS. Spark uses SparkContext to get access to a cluster. We provide the master node IP, the name of the application, the SparkHome and the jar path to SparkContext. We can use SparkContext to get a RDD and read files on HDFS. In the driver program, we need to read data from file on HDFS to get the attributes and values put into Dataframes.

We try to compare the approaches with rule mining approaches, AMIE+ [26] , Exception Enriched Rule Mining (only for EERM) [5] and HORNCONCERTO. The results of the comparisons for the other systems are taken from the work [27]. The systems are not very comparable as they are adapted and tested for different system environments. But since the implementation for both the systems are not available in Spark we draw this comparison from the existing results to depict the advantage of using the Spark based cluster environment for rule mining. Moreover, the original work [5] mentions the runtime for the algorithm varying majorly based on the size and distribution of the predicates in the KG. All the approaches require only a triple dataset as input and all the support and confidence values were taken same as the original works.

6.1.2 Preliminary Results

Runtime and Number of Rules

Dataset	Parser & FP-Growth	EWS Calculation	Fact Generation
WORDNET	39 sec	43 sec	1 min 30 sec
FREEBASE	53 sec	1 min 03 sec	1 min 54 sec
IMDB	1 min 18 sec	5 min 27 sec	4 hour 24 min
YAGO	1 min 52 sec	6 min 08 sec	2 hour 18 min

Table 6.2: Runtime EERM

The table 6.2 shows the runtime of the EERM. The rumtime depicted are the total time of the entire pipeline for the EERM as shown in figure 4.2, i.e., from horn rule mining, exception mining to generation of new facts. The Spark jobs runtime depend on network congestion between the nodes in the cluster as well as the HDFS, we averaged the runtime between three different iteration of the job. These figure include the read and the write time from and to different partitions of data on the HDFS cluster.

For WordNet, eerm produced more rules when compared to the HornConcerto which HornConcerto produced 365 Horn Rules in 12 sec and Amie+ produces 151 rules in 20 sec. The more runtime in case of small dataset can be explained because of the Spark initialization overhead. In case Freebase, eerm produces higher number of rules in much less time (2 hour 1 min and 2 hour 45 min for HornConcerto and Amie+, respectively).

The current runtimes are based on a distributed implementation of the algorithm combined with strategic caching and partitioning of data according to the algorithm. The difference between the run time of the rule mining and fact generation is due to the planning done within Spark in order to mine the exceptions. The optimized plans and the resultant jobs required to mine the exceptions requires much less shuffling of data and hence results in a faster throughput. The relatively larger fact generation time is a result of shuffling caused within Spark data structures due to different joins between the original KB.

Dataset	#Horn Rules	#ews	#Total
WordNet	169	208	377
Freebase	11,117	5,564	16,681
imdb	74,696	48,838	123,534
yago	2,715	1630	4,345

Table 6.3: Number of Rules of EERM

The table 6.3 shows the number of rules.We observed that FP-Growth produced a huge number of candidates for the ews calculation step. The higher number of horn rules in imdb as compared to the yago is due to the fact that imdb is a more connected dataset. Most of the rules with ews have a confidence score higher than 0.7 for imdb while for yago confidence for major number of rules with ews is 0.8 - 3.1.

Quality Based

In order to evaluate the quality of the rules, we evaluated the rules by a group of people who are not involved in this project. 20 rules were randomly chosen from yago and imdb and was passed on to 36 reviewers (apart from the author) as forms for their reviews. The form consisted of a shuffled set of

Rules	**Votes**
isAPerson(X) ⋀ isMarriedToTom(X) ⋀ ¬ hasGenderMale(X) ⟶ hasGenderFemale(X)	97.2%
createdBy21CenturyAmericanActress (X) ⋀ ¬ createdByAmericanJazzSingers(X) ⋀ createdByAmericanPopSingers (X) ⋀ createdByAmericanFilmActors (X) ⟶ createdByAmericanSingers (X)	94.4%
isLocatedInTurkey (X) ⋀ ¬ isLocatedInCilicia(X)⋀ isLocatedInAdanaProvince (X) ⟶ isTurkishAdministrativeDistrict (X)	91.7%
producedInUSA (X) ⋀ directedByAmericanScreenWriters (X) ⋀ ¬ createdByYeddishTheaterPerformers(X) ⟶ directedByAmericanPeople (X)	91.7%

Table 6.4: Top 4 rules from Exception Enriched Rule Mining (EERM)

yes/no response to the question *"Whether the rules made sense to you?"* for each rule. We shortened the URI and removed the namespaces for a easier understanding of the reviewers. Around 80% of the rules in the survey got a positive vote (yes) by at least 70% of the reviewers. The top four rules as agreed upon by reviewers is shown in Table 6.4. There were presence of some rules with very high confidence which did not make sense to a human reader. One such example is

$$\textsc{wasBornInUnitedStates}(X) \bigwedge \textsc{diedIn1977}(X) \bigwedge \\ \neg\textsc{diedinNewYorkCity}(X) \longrightarrow \textsc{diedInUSA}(X)$$

This rule has a surprisingly high confidence of 7.3, however, it is the lowest voted rule by reviewers. Presence of such rules, steers the future work in the direction of adding some sort of semantic attribute to the rule mining process or come up with some other evaluation methodology for the same.

We also observed the presence some tautologies i.e., the rules where the head is also present as a part of the body, as shown below. This is the result of presence of multiple namespace for the same operator in the datasets.

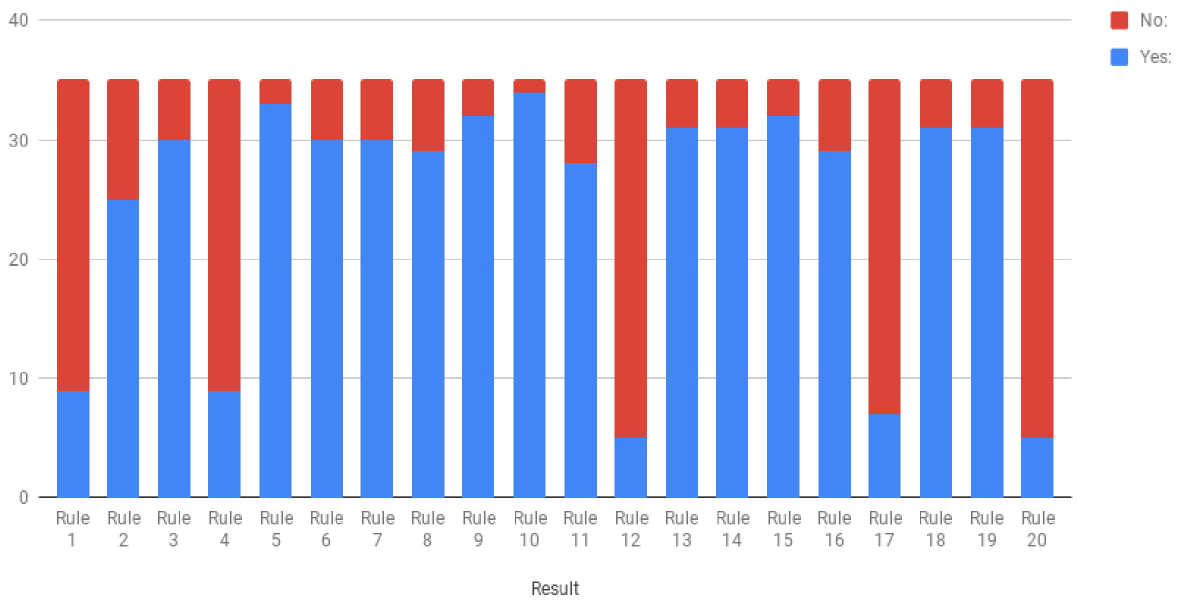

Figure 6.1: Agreement On Rules

$$\textsc{wasBornInUnitedStates}(X) \bigwedge \textsc{isAffiliatedToNewYorkUniversity}(X) \bigwedge \neg\textsc{wasBornInNorthKorea}(X) \longrightarrow \textsc{wasBornInUnitedStates}(X)$$

The agreement on all the rules reviewed is shown in figure 6.1. The numbering of the rules is done as follows:

1. directedByAmericanScreenWriter (X) $\bigwedge$ directedByAmericanProducers (X) $\bigwedge$ directedByAmericanPeople(X) $\bigwedge$ $\neg$ directedByAmericanActors(X) $\longrightarrow$ directedbyAmericanDirectors (X)
2. isADocument (X) $\bigwedge$ $\neg$ isAText(X) $\longrightarrow$ isAnArticle (X)
3. createdByWomen (X) $\bigwedge$ createdByAmericanSingers (X) $\bigwedge$ createdByAmericanPopSinger(X) $\bigwedge$ $\neg$ victimsOfAviationAccident(X) $\longrightarrow$ createdByLivingPeople (X)
4. createdByWomen (X) $\bigwedge$ createdByAmericanSingers(X) $\bigwedge$ createdByAmericanPopSinger(X) $\bigwedge$ $\neg$ createdByHipHopMusicians(X) $\longrightarrow$ createdByLivingPeople (X)

5. createdBy21CenturyAmericanActress (X) $\bigwedge$ createdByAmericanFilmActors(X) $\bigwedge$ createdByAmericanPopSinger(X) $\bigwedge$ $\neg$ createdByAmericanJazzSingers(X)
$\longrightarrow$ createdBy21CenturyAmericanSingers(X)

6. createdByWomen (X) $\bigwedge$ createdByAmericanPopSinger(X) $\bigwedge$ $\neg$ createdBy21CenturyAmericanActress(X)
$\longrightarrow$ createdBy21CenturyAmericanSingers(X)

7. createdByLivingPeople (X) $\bigwedge$ createdByAmericanScreenWriter(X) $\bigwedge$ createdByAmericanTelivisionWriters(X) $\bigwedge$ $\neg$ createdByPeopleofBritishColombia(X)
$\longrightarrow$ createdByAmericanTelivisionProducers(X)

8. createdByLivingPeople (X) $\bigwedge$ createdByAmericanScreenWriter(X) $\bigwedge$ createdByAmericanTelivisionWriters(X) $\bigwedge$ $\neg$ createdByPeoplefromSantaMonicaCarlifornia(X)
$\longrightarrow$ createdByAmericanTelivisionProducers(X)

9. isLocatedinTurkey (X) $\bigwedge$ isLocatedinAdnanaProvince(X) $\bigwedge$ $\neg$ isLocatedinCilicia(X)
$\longrightarrow$ isTurkishAdministrativedistrict(X)

10. isAPerson (X) $\bigwedge$ isMarriedToTom(X) $\bigwedge$ $\neg$ hasGenderMale(X)
$\longrightarrow$ hasGenderFemale(X)

11. isLocatedInSwitzerland (X) $\bigwedge$ isAMountain(X) $\bigwedge$ $\neg$ isLocatedinCantonOfBerny(X)
$\longrightarrow$ isLocatedinGraubnden(X)

12. wasBornInUnitedStates (X) $\bigwedge$ hasGenderMale(X) $\bigwedge$ isAPerson(X) $\bigwedge$ $\neg$ wasBornonDate194(X) $\longrightarrow$ hasWonPrizePurpleHart (X)

13. wasBornInUnitedStates (X) $\bigwedge$ isAffiliatedToNewYorkUniversity(X) $\bigwedge$ $\neg$ wasBornInSouthKorea(X) $\longrightarrow$ wasBornInUnitedStates (X)

14. wasBornOnDate (X) $\bigwedge$ isAffiliatedToWorkingMensParty(X) $\bigwedge$ $\neg$ hasGenderFemale(X) $\longrightarrow$ hasGenderMale (X)

15. producedInUSA (X) $\bigwedge$ directedByAmericanScreenWriters (X) $\bigwedge$ $\neg$ createdByYeddishTheaterPerformers(X) $\longrightarrow$ directedByAmericanPeople (X)

16. isAFotballPlayer (X) $\bigwedge$ wasBorninPoland (X) $\bigwedge$ $\neg$ diedinNetherlands(X) $\longrightarrow$ playsForLAGalaxy (X)

17. directedbyAmericanPeople (X) $\bigwedge$ $\neg$ directedByShowDogUniversalMusicArtists(X) $\longrightarrow$ directedByWesternGenreFilmDirector(X)

18. createdByLivingPeople (X) $\bigwedge$ createdByActors (X) $\bigwedge$ createdBy20thCenturyActors (X) $\bigwedge$ $\neg$ createdByEnglishActors(X) $\longrightarrow$ createdByAmericanActors (X)

19. wasBorninUK (X) $\bigwedge$ hasGenderMale (X) $\bigwedge$ isAffiliatedToManchesterUnited (X) $\bigwedge$ $\neg$ isAffiliatedToBasketballTeam(X) $\longrightarrow$ playsforEnglishSoccerTeam (X)

20. wasBorninUnitedStates (X) $\bigwedge$ diedin1977 (X) $\bigwedge$ $\neg$ diedInNewYorkCity(X) $\longrightarrow$ diedInUSA (X)

Link Prediction Based Quality Check

We tested the affect of EWS rules along with HornConcerto rules for link prediction evaluation. We use the same setting as HornConcerto [27]. The datasets are split into training, validation, and test set. For each triple in the test set, corrupted triples are generated by altering subjects and objects. Each triple has its confidence measure from either the HornConcerto or EERM. The probability is computed by taking the maximum value of confidence for triple as the intution is that probability of the missing link is proportional to the confidence scores of the rules involving its predicate. The *mean reciprocal rank* is obtained by inverting the average rank (by descending probability) of a correct triple among the corrupted ones. *Hits@N* measures the percentage of cases where a correct triple ranks in the top-N triples. While the original results of the system is available in the original paper, we draw the table 6.5 to show the improvement in the different metrics for HornConcerto. For WordNet, we see an improvement in the for both MRR and Hits@N results. The results are statistically not much higher and are in initial stages at the time of this work. In the future, we would want to tune the parameters for HornConcerto and plugin other state of the art approaches for this evaluation too.

Approach	MRR	Hits@1	Hits@3	Hits@10
HornConcerto	0.970908929524	0.969	0.9726	0.9736
HornConcerto + EERM	**0.973084087136**	**0.9728**	**0.9732**	**0.974**

Table 6.5: Link Prediction Results for WordNet

6.2 Decision Tree Rule Mining Results

6.2.1 Experimental Setup

Table 6.6: Datasets used in the evaluation

Dataset	# triples	Size of LIBSVM Data
WordNet (WN18)	3K	21MB
Freebase	3K	21.2MB
IMDb	25K	211 MB
YAGO	40K	300 MB

We faced network issues while running the MLLib library within the cluster provided which had a similar setup as described for EERM. The affect of this network resulted in problems in braodcasting of small parameters variables in Decision Tree implementation and hence the Spark jobs pausing for hours.

Since the issue couldn't be fixed in time for this work the experimental setup for the Decision Tree had to be changed and limited to standalone setup on a single 8GB 2,9 Ghz 4 core processor.

We took smaller version of the datasets to be able to fit on a single machine with the provided specification. The choice of the KBs allows us to evaluate our method's performance on both general-purpose and domain-specific KBs. The table 6.6 shows the number of triples taken from each of the dataset and corresponding size of the LIBSVM dataset. We observe that the size of the LIBSVM dataset grew with the increase in the number of triples as the number of files is equal to number of operators and size of each file contains entries equal to the number of triples.

Some preparations are made before executing the driver program. YARN and Spark environment needs to be deployed, and the training data should have been put on local HDFS setup. Spark uses SparkContext to get access to a cluster. We provide the master node IP (local system), the name of the application and the jar path to SparkContext. We use SparkContext to get a RDD and read files on the HDFS. In the driver program, we need to read data from file on hdfs to get the attributes and values put into Dataframes.The approaches require only triple dataset as input.

6.2.2 Preliminary Results

Runtime and Number of Rules

Dataset	LIBSVM Dataset Creator	DT Miner
WordNet	31 sec	4 min 12 sec
Freebase	36 sec	3 min 24 sec
imdb	3 min 27 sec	19 min 27 sec
yago	7 min 12 sec	34 min 18 sec

Table 6.7: Runtime DTRM

The table 6.7 shows the runtime of the DTRM. The runtime shown here are the total time of the rule mining for the DTRM. These figure include the read and the write time of data on the file system for each iteration. The run time presented are not in the same environment as the others and hence cannot be directly compared. The main intention behind the results is the proving the concept of a rule miner that can mine all sort of rules with one Spark job submission.

The dataset creator is a python based parallel implementation of the writer that utilizes all possible cores to write the files. The increase in time is directly proportional to the number of files. With increase in cores on larger servers the time may not increase in the same proportion. Each decision tree takes approximately 0.07 - 0.12 seconds to be built in the standalone system. The number of iteration i.e., based on the number of LIBSVM files also has a proportional effect on the total time. The table 6.8 shows the number of rules. As seen in EERM the less connected datasets produce less rules with high confidence. In our case all the rules obtained have a confidence of 1 as there is no pruning implemented on the tree.

Although the run time of the decision tree miner look reasonable, the size of the LIBSVM dataset is increasingly alarming. For the entire Wordnet and Freebase dataset the size of the LIBSVM data is close to 437 GB. With such a huge amount of data comes additional constraints such as the cluster size, memory along with the runtime of Spark jobs. Moreover, writing the entire

Dataset	#Rules
WordNet	67
Freebase	264
imdb	714
yago	143

Table 6.8: Number of Rules of DTRM

dataset on the HDFS implies the number reads from HDFS to be at least #operators times, which is not feasible.

Quality Based

In order to evaluate the quality of the rules, I manually went through the files for imdb and yago to check the quality of the rules and evaluate it based on the similar criteria as for the EERM rules. However, in this case the number of reviewers is limited to the author. Even though the size of dataset is small to generalize, some of the interesting rules that are mined which cover overall aspects of quality aspect of rules mined are as follows:

1. directedByLivingPeople (X) $\bigwedge$ directedByAmericanActors(X) $\longrightarrow$ directedbyAmericanPeople (X)
2. createdByAmericanWriters(X) $\longrightarrow$ createdByAmericanPeople (X)
3. createdInUSA (X) $\bigwedge$ createdByAmericanScreenWriters$\bigwedge$ $\neg$ createdByYeddishTheaterPerformers(X) $\longrightarrow$ createdByAmericanPeople (X)
4. createdByAmericanPeople (X) $\bigwedge$ createdByAmericanScreenWriters$\bigwedge$ createdByAmericanFilmDirectors(X) $\longrightarrow$ directedByAmericanFilmProducers (X)
5. bornInSantaClaraCounty(X) $\longrightarrow$ bornInSantaClaraCountyCalifornia(X)
6. bornInNewYorkMetropolitanArea(X) $\bigwedge$ bornInProvinceNewYork(X) $\longrightarrow$ bornInUnitedStates(X)
7. locatedInNorfolk(X) $\longrightarrow$ locatedInNorfolkEngland(X)

8. locatedInWarmianMasurianVoivodeship(X) $\bigwedge$ $\neg$ locatedInDziałdowoCounty(X) $\bigwedge$ $\neg$ locatedInGołdapCounty(X) $\longrightarrow$ locatedInWarmiaMasuria(X)

9. directedbyAmericanPeople (X) $\bigwedge$ $\neg$ directedByShowDogUniversalMusicArtists(X) $\longrightarrow$ directedByWesternGenreFilmDirector(X)

10. locatedInPoland(X) $\bigwedge$ locatedInWarmiaMasuria(X) $\bigwedge$ $\neg$ locatedInEurope(X) $\longrightarrow$ locatedInWarmianMasurianVoivodeship(X)

Most of the rules were valid and contained both positive and negative body elements as we expected. Also the size of the rules are not limited and they are of varying sizes because of the variable number of body elements. The size of head is limited to one predicate. However, we did get some very interesting cases. The rule like Rule 5 and Rule 7 are quite prominent in the results. They are tautologies i.e., the operator in the head also occur in the body. The reason behind this is slight difference in URI or namespace of the resource i.e., both operator are not exactly unique. This is similar to the presence of tautologies in EERM results. We also had some very poor quality rules (Rule 9 and 10) which are not valid in any scenario. The reason behind such low quality rule could be because of absence of any other corresponding application with respect to the predicates in the dataset which was used for testing. In the future, if these rules exist after the simulation over the complete KBs, a rule revision methodologies would have to be put in place.

Chapter 7

Conclusions and Future Work

7.0.1 Conclusion

We presented a Exception Enriched Rule Mining approach in a distributed cluster based environment. The EERM, first learns a set of Horn rules from the KB, and then revises them by adding negated atoms into their bodies with the goal of improving the quality of a rule set for data prediction. We use a series of joins to calculate the exceptions for the rules. The algorithm proposed is different from the original iterative approach. It is adopted and exploits the computation power of Spark in a cluster. It showed great performance for major datasets in terms of runtime. However, the fact generation module took quite some time to generate facts as the most optimized plan requires a lot of shuffling data and hence more processing time. Moreover the quality of rules evaluation both via human reviewers and link prediction showed positive results for EERM.

We also proposed a novel rule mining approach which uses decision tree rule mining. This process takes triples, creates predicates along with corresponding ids. The transformed KB is used to create the LIBSVM dataset and run the decision tree implementation of Spark to generate rules from the tree. These rules can have both positive and negative body elements i.e., non monotonic rules based on decision trees created from the KB. Moreover, either of the positive and negative part of the body can be absent. This methodology had serious concern with the size of the input dataset, however, the quality of rules were quite good.

7.0.2 Future Work

Both these work will be a part of SANSA [39, 46], an open source *data flow processing engine* for performing distributed computation over large-scale

RDF datasets. More information about SANSA is available at `https://github.com/SANSA-Stack`.

There are various directions for improvement. Firstly, this work is limited by Spark's architecture and implementation joins and groubBy and hence the EERM algorithm modules has a scope for optimization. Currently, the implementation consists of a series of dataframe joins which involves shuffling of entire data multiple times. We are currently able to get results by optimizing the number of partitions. An interesting approach would be to replace some of the joins with Scala specific features like *foldLeft* which might reduce the number of shuffles. Different test on making the algorithm more read and write oriented distributed over a series of Spark job can also be an interesting approach. I believe that a further improvement in the time of the algorithm can be achieved by implementation of an index small in size that can be easily broadcasted into the cluster resulting in a higher performance in joins.

Secondly, the evaluation of the work can be built upon to create a better check for the quality of rules. The quality of rules could be verified from the KB related text or corpora. We would also like to do a more tests with different

Thirdly, the test for the decision tree rule miner could be run on a larger cluster once the network issues within the cluster is fixed and the performance could be analyzed. Pruning methodologies to in corporate more rules with slightly less confidence scores. Optimization of joins to be able to produce a better id to operator map for the rules as mentioned previously. Finally, a more complex rules with multiple heads can be mined by joining multiple rules.

Bibliography

[1] F. M. Suchanek and N. Preda, “Semantic culturomics,” *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1215–1218, 2014.

[2] O. PATHFINDING, M. FIRST-ORDER, and K. F. L. K. BASES, “Probkb,”

[3] Z. Wang and J. Li, “Rdf2rules: learning rules from rdf knowledge bases by mining frequent predicate cycles,” *arXiv preprint arXiv:1512.07734*, 2015.

[4] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek, “Fast rule mining in ontological knowledge bases with amie +,” *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 24, no. 6, pp. 707–730, 2015.

[5] M. H. Gad-Elrab, D. Stepanova, J. Urbani, and G. Weikum, “Exception-enriched rule learning from knowledge graphs,” in *International Semantic Web Conference*, pp. 234–251, Springer, 2016.

[6] J. Dean and S. Ghemawat, “Mapreduce: a flexible data processing tool,” *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *ACM SIGOPS operating systems review*, vol. 41, pp. 59–72, ACM, 2007.

[8] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, “Map-reduce-merge: simplified relational data processing on large clusters,” in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 1029–1040, ACM, 2007.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets.,” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[10] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, “A review of relational machine learning for knowledge graphs,” *Proceedings of the IEEE*, vol. 104, no. 1, pp. 11–33, 2016.

[11] R. Agrawal, T. Imieliński, and A. Swami, “Mining association rules between sets of items in large databases,” in *Acm sigmod record*, vol. 22, pp. 207–216, ACM, 1993.

[12] D. Taniar, W. Rahayu, V. Lee, and O. Daly, “Exception rules in association rule mining,” *Applied Mathematics and Computation*, vol. 205, no. 2, pp. 735–750, 2008.

[13] Z. Abedjan and F. Naumann, “Improving rdf data through association rule mining,” *Datenbank-Spektrum*, vol. 13, no. 2, pp. 111–120, 2013.

[14] J. JÓzefowska, A. Ławrynowicz, and T. Łukaszewski, “The role of semantics in mining frequent patterns from knowledge bases in description logics with rules,” *Theory and Practice of Logic Programming*, vol. 10, no. 3, pp. 251–289, 2010.

[15] Z. Abedjan, J. Lorey, and F. Naumann, “Reconciling ontologies and the web of data,” in *Proceedings of the 21st ACM international conference on Information and knowledge management*, pp. 1532–1536, ACM, 2012.

[16] Z. Abedjan and F. Naumann, “Synonym analysis for predicate expansion,” in *Extended Semantic Web Conference*, pp. 140–154, Springer, 2013.

[17] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok, “Frequent subtree mining–an overview,” *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 161–198, 2005.

[18] M. Kuramochi and G. Karypis, “Frequent subgraph discovery,” in *Data Mining, 2001. ICDM 2001, Proceedings IEEE international conference on*, pp. 313–320, IEEE, 2001.

[19] S. A. Macskassy, F. Provost, and F. Provost, “Netkit-srl: A toolkit for network learning and inference,” in *Proceeding of the NAACSOS Conference*, 2005.

[20] D. Jain and M. Beetz, “Soft evidential update via markov chain monte carlo inference,” in *Annual Conference on Artificial Intelligence*, pp. 280–290, Springer, 2010.

[21] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, J. Wang, and P. Domingos, "The alchemy system for statistical relational {AI}," 2009.

[22] J. R. Quinlan, "Learning logical definitions from relations," *Machine learning*, vol. 5, no. 3, pp. 239–266, 1990.

[23] S. Muggleton, "Inverse entailment and progol," *New generation computing*, vol. 13, no. 3-4, pp. 245–286, 1995.

[24] S. Muggleton, "Learning from positive data," in *International Conference on Inductive Logic Programming*, pp. 358–376, Springer, 1996.

[25] S. Schoenmackers, O. Etzioni, D. S. Weld, and J. Davis, "Learning first-order horn clauses from web text," in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pp. 1088–1098, Association for Computational Linguistics, 2010.

[26] L. A. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek, "Amie: association rule mining under incomplete evidence in ontological knowledge bases," in *Proceedings of the 22nd international conference on World Wide Web*, pp. 413–422, ACM, 2013.

[27] T. Soru, A. Valdestilhas, E. Marx, and A.-C. N. Ngomo, "Beyond markov logic: Efficient mining of prediction rules in large graphs," *arXiv preprint arXiv:1802.03638*, 2018.

[28] E. Miller, "An introduction to the resource description framework," *Bulletin of the American Society for Information Science and Technology*, vol. 25, no. 1, pp. 15–19, 1998.

[29] B. McBride, "The resource description framework (rdf) and its vocabulary description language rdfs," in *Handbook on ontologies*, pp. 51–65, Springer, 2004.

[30] D. L. McGuinness, F. Van Harmelen, *et al.*, "Owl web ontology language overview," *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.

[31] J. Fürnkranz, D. Gamberger, and N. Lavrač, "Rule learning in a nutshell," in *Foundations of Rule Learning*, pp. 19–55, Springer, 2012.

[32] J. W. Lloyd, *Foundations of logic programming.* Springer Science & Business Media, 2012.

[33] D. Borthakur *et al.*, “Hdfs architecture guide,” *Hadoop Apache Project*, vol. 53, pp. 1–13, 2008.

[34] AgiData, “Apache spark cluster managers: Yarn, mesos, or standalone?,” 2015.

[35] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, *et al.*, “Spark sql: Relational data processing in spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, ACM, 2015.

[36] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “Graphx: Graph processing in a distributed dataflow framework.,” in *OSDI*, vol. 14, pp. 599–613, 2014.

[37] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, *et al.*, “Mllib: Machine learning in apache spark,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[38] I. Ermilov, J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, S. Bin, N. Chakraborty, H. Petzka, M. Saleem, *et al.*, “The tale of sansa spark.,” in *International Semantic Web Conference (Posters, Demos & Industry Tracks)*, 2017.

[39] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. N. Ngonga, and H. Jabeen, “Distributed semantic analytics using the sansa stack,” in *Proceedings of 16th International Semantic Web Conference - Resources Track (ISWC’2017)*, 2017.

[40] J. Shafer, R. Agrawal, and M. Mehta, “Sprint: A scalable parallel classi er for data mining,” in *Proc. 1996 Int. Conf. Very Large Data Bases*, pp. 544–555, Citeseer, 1996.

[41] J. Gehrke, R. Ramakrishnan, and V. Ganti, “Rainforest-a framework for fast decision tree construction of large datasets,” in *VLDB*, vol. 98, pp. 416–427, 1998.

[42] S. R. Safavian and D. Landgrebe, “A survey of decision tree classifier methodology,” *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[43] M. N. Anyanwu and S. G. Shiva, "Comparative analysis of serial decision tree classification algorithms," *International Journal of Computer Science and Security*, vol. 3, no. 3, pp. 230–240, 2009.

[44] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, and V. S. Tseng, "Spmf: a java open-source pattern mining library," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3389–3393, 2014.

[45] C. Borgelt, "An implementation of the fp-growth algorithm," in *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pp. 1–5, ACM, 2005.

[46] I. Ermilov, J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, S. Bin, N. Chakraborty, H. Petzka, M. Saleem, A.-C. N. Ngonga, and H. Jabeen, "The Tale of Sansa Spark," in *Proceedings of 16th International Semantic Web Conference, Poster & Demos*, 2017.

[47] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: parallel fp-growth for query recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*, pp. 107–114, ACM, 2008.

[48] S. Muggleton, C. Feng, *et al.*, *Efficient induction of logic programs.* Citeseer, 1990.

[49] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[50] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[51] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Advances in neural information processing systems*, pp. 2787–2795, 2013.