Bachelorthesis

Association Rule Mining of Linked Data Using Apache Spark

Nathan, Theresa
Matrikel-Nr: 2585148

University of Bonn

Version of: September 4, 2018

Supervisor : Professor Dr. Jens Lehmann

Table of Contents

Abstract. In the modern world the amount of big data that is produced every day has rapidly grown. RDF knowledge bases are one way to publish structured data. But these knowledge bases are often not complete or contain false data. In my bachelor thesis I propose a way to complete RDF knowledge bases. AMIESpark uses the approach of the AMIE+ team to mine association rules. SPARK data structures like RDDs and DataFrames are used to calculate the confidence threshold, which is used to evaluate the usefulness of a rule, while operations of the AMIE+ algorithm manage the building and output of the rules. After testing different approaches to optimize runtime, we come to the conclusion that our algorithm is not the most efficient way to mine association rules.

1 Introduction

In the modern world the amount of big data that is produced every day has rapidly grown. One way to structure that data are RDF knowledge bases (KB). RDF (Resource Description Framework) is a model used in the web to link data by defining relationships between things. One problem with big data and RDF knowledge bases is that they are often incomplete, or contain false data [1]. The AMIE+ team tried to solve this problem by writing an association rule mining algorithm to generate rules that complete RDF knowledge bases and identify false entries. [2] Using RDF knowledge bases makes it difficult to use traditional mining approaches suited for tabular data formats, because it operates on an open world assumption (OWA). "Under the OWA, a statement that is not contained in the KB is not necessarily false; it is just unknown" [3].

1.1 Motivation

As the volume of big data is growing every day, the research on ways to process this data is extremely important. Seeing the approach of the AMIE+ team, where java is used, the idea of using SPARK instead, in hopes of improving the runtime, is worth exploring. SPARK is optimized for working with data sizes AMIE+ has problems with, so using it should make this possible. Also, having access to the newly developed SANSA stack and integrating the AMIESpark program into it, widens the possibilities of doing research on that approach. The decision of trying to improve AMIE+ is based on its effectiveness. AMIE+ is able to mine rules without explicit counter examples or additional facts [3]. Furthermore it can run on KB's with millions of facts and return results "in a matter of hours or minutes" [3]. In their paper, the AMIE+ team describes pruning methods to get the rate of rules that are correct up to 70 percent. They also compare the run time and success rate of AMIE+ to other state of the art algorithms, so its effectiveness is put into perspective.

1.2 Objective of the work

As described in the previous paragraph, SPARK as a tool to implement the AMIE+ algorithm and maybe improve the runtime is worth exploring. Also,

the exploration of SPARK native data structures like RDDs, DataFrames and Datasets are of great interest, as well as trying different mechanics to shorten the runtime like saving intermediate results as parquet files and different configuration options, when submitting SPARK applications or initializing the SparkContext. In summary, the objective of the work is to choose the best data structure, tools and configurations to implement the AMIE+ algorithm with SPARK, while still making use of its inherent good qualities.

1.3 Related Work

There are publications that mine rules similarly to AMIE+ but do not meet the requirements of AMIE+. WARMER, QuickFOIL[3] and ALEPH for example are also Rule Mining systems, but they are not able to operate on a large KB on an open world assumption. And because AMIE+ aims to mine horn rules, various other approaches are not sufficient [2]. [4] for instance uses "rule mining to generate the schema or taxonomy of a KB".
[5]and [6] are candidates as well, but [5] is an algorithm that uses ontologies as well as schema data and [6] uses the T-Box, so they are both not fit to use for AMIE+, because it takes only the A-Box into account. But a future task could be to alter AMIE+ in such a way that you can use ontologies with OWL as well, like in [5]. Or to make use of the T-Box like suggested in [6] to derive information from the semantic web, instead of using available RDF knowledge bases. In [7] a rule mining algorithm is used to find OWL axioms between two atoms, for example the inverse to an atom. But the approach of AMIE+ yields far more information so that AMIE+ is my preferred choice. Another popular approach is to mine rules in a "generalized form" to collect information from RDF KB's [8]. Generalized rules are RDF atoms which are a generalization of a group of other atoms. [8] mines generalized rules with the help of Apriori-based Association Rule Mining, which is similar to WARMER and thus not able to operate on large KB's. [9] operates under the OWA like AMIE+, but it mines cycles inside the RDF Graph from which rules are mined later, while AMIE+ mines rules from a RDF KB. Their approach can be used later on to try to use the T-box to produce more accurate rules. ScaLeKB [10] can be of great use while optimizing AMIESpark, as it describes efficient join processing and performance tuning through partitioning.

1.4 Structure of the Thesis

In the first section the aim, motivation and relation of other work to the thesis is described. In the second section important vocabulary is defined and explained, revisiting the work of the AMIE+ team , [3], as AMIESpark uses all main operations, confidence thresholds and its structure and implements them anew using SPARK native data structures and functions. They are also defined in this section, as well as SPARK itself. A description of the resulting challenges is given at this point. Section three concentrates on AMIESpark as an algorithm and on different approaches when implementing it to shorten its runtime. The forth section

is the evaluation of the implementation choices and SPARK as a implementation tool for AMIE+. Here the conclusion of this evaluation is drawn. Section five summarizes the whole thesis and gives an outlook on other approaches to implement the algorithm.

2 Preliminaries

2.1 RDF KBs

Like AMIE+, AMIESpark uses RDF KBs to operate. As described in the preliminaries of [3], RDF KBs are lists of facts, where facts are denoted as a relationship between a subject and an object. Facts can be written as *r(x,y)* or as a triple (x,r,y) [3]. r is the relationship, while x is the subject and y is the object. One example for a fact in a KB could be *isChildOf(Dave,James)* or *(Dave,isChildOf,James)*. Knowledge Bases consist of an A-Box, which are statements about the relationship of one entity to another, and a T-Box, which defines concepts and properties for an entity. But AMIE+ as well as AMIESpark only use the A-Box for rule mining [3].

2.2 Rules

In order to make predictions based on the Knowledge Base, *rules* have to be build by the Algorithm. These rules consist of atoms, which are facts with the subject and object position substituted with variables. The rule can be divided in a head, one atom, and a body, a list of atoms. Like in [3] the rule with the head as *r(x,y)* and the body as $\{B_1, \ldots, B_n\}$ is an implication of the form

$$B_1 \wedge B_2 \wedge \ldots \wedge B_n \Rightarrow r(x, y)$$

which can be written as $\overrightarrow{B} \Rightarrow r(x, y)$ [3]. An *instantiation* of a rule has constants in the place of the variables and is also called positive example, if the instantiated rule is build out of facts of the KB. As mentioned earlier, rules are build to make *predictions*. These can be made for the head of the rule by finding instantiations of the body in the KB. If we have the rule $livesIn(x, y) \wedge isMarriedTo(x, z) \Rightarrow livesIn(z, y)$ and instantiate the body with facts of the KB *livesIn(Steve,London)* $\wedge$ *isMarriedTo(Steve,Sara)* the new fact *livesIn(Sara,London)* can be derived.
To limit the search space and stop the algorithm from mining rules with atoms that have no connection, the atoms in a rule have to be *connected*. This means that every atom has to be connected transitively to every other atom of the rule [3]. This connection is guaranteed by the *danglingAtom Operator* and the *closingAtom Operator*, which will be described in detail later on. To exclude rules that are only able to predict the existence of a fact, the rules that are put into the output list also have to be *closed*. A rule is *closed* when all variables appear at least twice in the rule [3]. Also, atoms with the same variable in subject and object position are not used to build rules as the search for connections between

different subjects and objects is of greater importance. But as seen in a previous example, AMIESpark allows the building of *recursive rules*, rules with the head relation in the body, in contrast to other ILP systems [3].

2.3 Confidence Measures

To decide which rules to refine or even output, the algorithm uses different thresholds. The Head coverage is a proportional version of the support [3], which is the number of positive examples for a rule, and is used as a threshold in the refinement process. The PCA Confidence is necessary to make sure the algorithm picks rules that make predictions that are new to the KB.

2.3.1 Support and Head Coverage As mentioned earlier, the support describes the number of instantiations of a rule. This interpretation of support was chosen with the goal of *monotonicity* in mind [3]. With the addition of new atoms and more constraints to the rule, the support decreases, which is an important property to be able to prune the rule. But the support alone is not enough to measure the precision of a rule. The Head Coverage is ratio of known true facts to the size of the head relation *size(r)* of the rule [3]. The size of the head relation is the number of positive examples that exist for the head of the rule.

$$hc(\overrightarrow{B} \Rightarrow r(x,y)) = \frac{support(\overrightarrow{B} \Rightarrow r(x,y))}{size(r)}$$

The head coverage of the example in Table 1 is $hc(livesIn(x,y) \wedge isMarriedTo(x,z) \Rightarrow livesIn(z,y)) = \frac{1}{6}$, as there are 6 facts in the column of the head atom and 1 positive example.

livesIn	isMarriedTo
(Steve,London)	(Steve,Sara)
(Peter,Paris)	(Peter,Sara)
(John,Paris)	(John,Sara)
(James,Berlin)	(Lisa,James)
(Sara,London)	
(Sonja, Rome)	

Table 1: Example table of facts for the rule $livesIn(x,y) \wedge isMarriedTo(x,z) \Rightarrow livesIn(z,y)$ with one positive example (support) and two negative examples

2.3.2 PCA Confidence The aim of AMIESpark is to find new rules that make it possible to predict new facts to complete the KB. To be able to do this,

negative examples have to be taken into account. AMIE+ does this by making use of the *Partial Completeness Assumption* (PCA). The PCA assumes that "if we know one y for a given x and r, then we know all y for that x and r" [3]. Through this assumption, negative examples can be generated. The PCA Confidence can be written as

$$pcaConf(\overrightarrow{B} \Rightarrow r(x,y)) = \frac{support(\overrightarrow{B} \Rightarrow r(x,y))}{\#(x,y):\exists z_1,z_2,...,z_m,y':\overrightarrow{B} \wedge r(x,y')}$$

In the example in Table 1 the PCA dictates that there is only one city for Seve, Sara, Peter John, James, and Sonja they are living in. $livesIn(Steve, London) \wedge isMarriedTo(Steve, Sara) \Rightarrow livesIn(Sara, London)$ is counted as a positive example. *livesIn(Peter,Paris)* $\wedge$ *isMarriedTo(Peter,Sara)* and *livesIn(John,Paris)* $\wedge$ *isMarriedTo(John,Sara)* could be bodies of two new positive rules, but then Sara would have to live in Paris. She is already linked to London so the two bodies are counted as negative examples. *livesIn(James,Berlin)* and *livesIn(Sonja,Rome)* are disregarded, as they have no connection to a subject of a positive rule and have not enough facts to which they could connect themselves to to be a positive example.

2.4 challenges in rule mining for RDF

To use semantic KBs and RDF schemata to mine rules yields a number of challenges. State of the art rule mining systems are Inductive Logic Programming (ILP) systems. "ILP approaches induce logical rules from ground facts" [3]. But ILP systems require negative examples to run properly, which are not part of semantic KBs. And RDF schemata are to weak to deduce negative statements from the semantic KB [3]. Also, the algorithm operates under the OWA, so "absent statements cannot serve as counter-evidence" [3]. In conclusion ILP approaches can not be used to mine rules from a semantic KB. The AMIE+ team solved the problem of the absence of negative examples by using the Partial Completeness Assumption (PCA) to estimate negative statements [3]. With that, the quality of rules can be computed, resulting in a high percentage of correct rules that are mined.

2.5 SPARK

Apache SPARK is a cluster computing system that is optimized for working with Big Data. Big Data are large amounts of data that can not be processed in conventional ways, because of their size and their complexity. A SPARK application is separated in a driver program and the executors. The driver program plans the work flow and divides it into jobs. These jobs trigger the executors, which are spread on the clusters nodes and execute the work that has to be done. A RDD (resilient distributed dataset) is one of SPARKs data structures, that makes it possible to run a program in parallel. To understand the architecture of SPARK, Fig. 1, getting to know RDDs is necessary. RDDs are partitioned over the cluster and are operated on in parallel [11]. They support the operations *actions* and

transformations. Actions lead to computation and return values, in contrast to transformations, which return a transformed RDD, but are only executed after an action's execution. This is called lazy evaluation. The lazy evaluation of transformations allows SPARK to see everything that has to be done with the RDDs data until an action takes place. Therefore SPARK can optimize this process. It will figure out the best way to get to the values required by the action, without materializing the intermediate results of the transformation.
In the following, the process of transforming and invoking an action on RDDs will be described. The process begins with invoking an action inside the SPARK application, which launches a job to fulfill it [12]. Then, an execution plan is derived by taking the transformations that transform the dataset the action depends on into consideration and assembles them into *stages*. "Each stage contains a sequence of transformations that can be completed without shuffling the full data" [13]. Shuffling means that data is reorganized across partitions, so that specific operations can be computed. The stages are divided into tasks that run on the same code but work on different parts of the data [12].

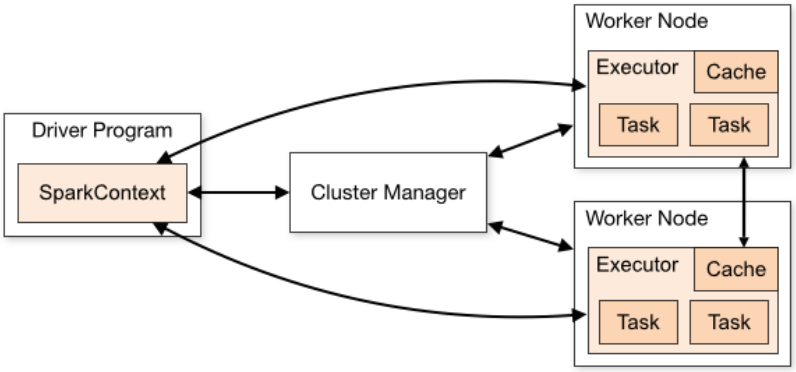

Fig. 1: SPARK architecture
source: `https://spark.apache.org/docs/latest/cluster-overview.html`

2.5.1 DataFrames and Datasets Datasets are distributed collections of data which are able to be optimized even more than RDDs, thanks to Spark SQL Catalyst optimizer [14]. Another advantage Datasets have, is their space and speed efficiency in comparison to RDDs. There are two APIs used in conjunction with Datasets. The untyped API in connection with Rows, Dataset[Row], which are also called DataFrames, and the strongly-typed API [15]. When using DataFrames or Datasets, it is possible to catch errors in compile time, a great advantage for developers. Because DataFrames are organized as tables, you are able to use SQL operations like aggregate, average, sum and join. The structuring like tables, the possibility of using joins, its advanced optimization and

speed informed the decision to use Dataframes as the main data structure for expensive computations in AMIESpark.

3 Approach

3.1 The Algorithm

To mine rules, AMIE+ uses RDF KBs, whose facts are divided in an A- and T-Box. AMIE+ only uses the A-Box, which contains instance data while it ignores the T-Box, which "is the subset of facts that defines classes, domains, ranges for predicates, and the class hierarchy" [3]. The algorithm, as you can see in Algorithm 1, takes a KB K, a threshold *minHC* on the head coverage of the mined rules, a maximum rule length *maxLen* and a minimal confidence threshold *minConf*, as input [3]. The way the algorithm works, is that it iterates until maximal length of rules (line 5). It takes a list of rules, the *iteration list*, which initially consists of all possible head atoms [3] (line 2), and iterates over it (line 9). All rules in the iteration list are *RuleContainers*. *RuleContainers* is the class that contains all important values for a rule like the *pcaConfidence*, *support*, its *parents* and the rule itself in the form of a list of its atoms. The first atom in that list is always the head, the rest is the body.
When the iteration list is first initialized, the head atoms are not only made into RuleContainers and added to the list, but also saved in the form of a DataFrame, where the subject, predicate and object form its columns. It contains all distinct facts of the KB and is called *facts table* or *dfTable*.
To build new rules out of these single atoms the algorithm adds new atoms to elements of the iteration list. This process is called refinement. In the first iteration, the rules are of size one. In the refinement process a second atom is added, so the new rule is of size two. In the second iteration, the rule candidates from the previous iteration that are saved in the *dataFrameRuleParts* RDD (Alg. 1, line 4), are turned into *RuleContainers* and, after a duplicate check, added to the new iteration list (line 6 to 8).
After building the iteration list, the algorithm iterates over it (line 9) and every element of the list is checked for whether it can be saved in the output list (line 11). To decide when to output a rule, the rule has to pass certain thresholds and criteria. The rule has to be closed, its confidence has to be higher than the *minConf*, which is done while generating the new rule, and improve confidence over all its parents [3]. A closed rule is a rule in which every variable appears twice or more. In the function *acceptedForOutput()* (Alg. 7) these thresholds are tested and if they are met, the rules are added to the output list (line 11 to 13). For checking if the confidence improved over all parents, not the standard confidence but the *PCAConfidence* is used. PCA stands for *Partial Completeness Assumption*. This assumption allow us to generate counter-examples in a way that is "less restrictive than the standard confidence" [3]. In AMIESpark the calculation of the *PCAConfidence* is done by dividing the support through the *PCABodySize*. The *PCABodySize* is the number of positive and negative examples.

Next, the element goes through a refinement process, if the rule (the parent) has less arguments than *maxLen*, which produces new rules (the children) (line 14, 15). The refinement process uses two operators to explore the search space in an efficient way. The first operator adds dangling atoms and the second adds closing atoms. Dangling atoms are atoms with a fresh variable as one of its arguments, while the other argument is a variable that already appeared in the rule (shared variables) and closed atoms are atoms that contain two shared variables [3]. The operators generate all possible new atoms to form a new rule and build these new rules as output candidates. Their corresponding support is also computed, as it is needed to check if it is able to pass the head coverage threshold. If they do not pass the threshold they are filtered out. When the Algorithm finishes, all rules in the output list are saved in a textfile.

Algorithm 1 Rule Mining

```
1: function ruleMining(KB K, minHC, maxLen, minConf)
2:     q = Rules of length one for first iteration of loop
3:     out = [ ]
4:     dataFrameRuleParts = null
5:     for i = 0 to this.maxLen − 1 do
6:         if i > 0 dataFrameRuleParts ≠ null then
7:             q = previously computed rules that are saved in dataFrameRuleParts
   made into RuleContainers
8:         end if
9:         for j = 0 to q.length − 1 do
10:            r = q(j)
11:            if AcceptedForOutput(r, outMap, minConf) then
12:                out.add(r)
13:            end if
14:            if length(r) < maxLen then
15:                dataFrameRuleParts = refine(r, dataFrameRuleParts)
16:            end if
17:        end for
18:    end for
19: end function
```

3.1.1 Refinement The refinement process is an important part of AMIES-park and also very time consuming. It adds wholly new rule candidates to the *dataFrameRuleParts* RDD that contains all the rules that will be made into the iteration list in the algorithms next iteration. The function *refine* is described in Algorithm 2 and takes the rule that has to be refined in the form of a *RuleContainer* and the *dataFrameRuleParts* RDD, which gets updated with every call of *refine*, as parameters. It returns the *dataFrameRuleParts* RDD, which becomes the new *dataFrameRuleParts* RDD for the next call of *refine* (Alg. 2 line 15). To refine the rule, the operators *addDanglingAtom* (Alg. 2 line 6) and *addClosedAtoms* (Alg. 2 line 8) are used. If the rule length is *maxLength - 1* only

the *addClosedAtoms* is used, because at this point there will be no further step to close the dangling atoms of this iteration. The results of both operations are DataFrames which contain all positive examples for all new rules that were build out of the new atoms and the rule. Both results are unified in line 9 to 13 of Alg. 2. The unification is made into a RDD. Then the results of each new atom are grouped together and counted by using the SPARK operation reduceByKey() (Alg. 2 line 14). The result of this is a RDD consisting of the new atom, the position of the rule, which is being refined, in the iteration list, and the number of positive examples. The position of the rule is needed as not the entire new rule with the new atom is saved in *dataFrameRuleParts* RDD, only the new atom. But to the whole rule is needed when building the *RuleContainers* in the next iteration. For that we have the position of the old rule in the old iteration list, which the new atom has to be attached to. To be able to limit the search space, intermediate results have to be filtered with the head coverage threshold. This happens by filtering out the atoms, whose example count is smaller than the number of facts for the head relation multiplied with minimal head coverage (*minHC*) (Alg. 2 line 15). This formula is the translation of the head coverage into an absolute support value [3]. Finally, the *dataFrameRuleParts* RDD is updated with the result of the filter and returned (Alg. 2 line 16).

Algorithm 2 Refinement

```
1: function REFINE(r, dataFrameRuleParts)
2:     OUT = dataFrameRuleParts
3:     tpAr = r.getListOfAtoms
4:     out = null
5:     if tpAr.length ≠ maxLen − 1 then
6:         out = kb.addDanglingAtom(r)
7:     end if
8:     temp = kb.addClosingAtom(r)
9:     if out == null then
10:        out = temp
11:    else
12:        out = out.unionAll(b)
13:    end if
14:    count = out is made into rdd and positive examples of all new found rules are
       counted with reduceByKey()
15:    o = count is filtered where (count.positiveExamplesCount ≥
       (relationSize(head) ∗ minHC)))
16:    return OUT.union(o)
17: end function
```

In the following, the operators *addDanglingAtom* and *addClosingAtom* will be described. Like explained earlier, the rules build with these two operators have to satisfy the head coverage threshold and for that the support is needed. Therefore a function to generate all positive examples is called inside the opera-

tors, which will also be described in the following. When the *addDanglingAtom* function is called, Tuples of a fresh variable and a variable that already exists in the rule are generated and saved in a list that is returned. If the rule is not closed, the fresh variable will be connected to the not connected variables. The Tuples that are generated for the rule $isMarriedTo(a, c) \Rightarrow livesIn(a, b)$ are {(c,d),(d,c)}. If the rule is closed, Tuples are build with every variable of the rule. The *addClosingAtom* function produces Tuples so that every open variable is closed. It does that by generating the most efficient Tuples for closing the rule. For the example the generated Tuples would be {(b,c),(c,b)}. If there is only one closed variable the Tuples will be formed between the open variable and all other variables [3]. And if the rule is already closed, all possible combinations of the variables are build. After the list of Tuples is generated, it is passed as *variablesNewAtom* as parameter to the function *countProjectionQueriesDF*, which is described in Algorithm 3.
The rules list of atoms, *tpAr* is also passed. This function, as mentioned earlier, will generate all positive examples for the newly build rule candidates. But before this can happen, the new atom and then the new rule have to be assembled. The new rule (the child) is build by adding the new atom to the rule that is being refined (the parent). The operator functions only send Tuples with the new variables, which will be the subject and object of the new atom, but the predicates of the new atom still have to be produced. For this, the Map *relationSize*, which had previously been computed by the algorithm, can be used. This contains all predicates of the facts table mapped to the number of their occurrences in the facts table(Alg. 3 line 10). So if the predicate *isMarriedTo* appears 5 times in the facts table, the Map looks like *Map(isMarriedTo → 5)*. This Map is build in the beginning, when the input file is being made into the first iteration table. Every relation is counted and saved into that Map in connection with its number of occurrence in the iteration table.
That Map and *variablesNewAtom* is iterated over (Alg. 3 line 11) to combine all Tuples, i_1 and i_2, with the current predicate, which is denoted by x_1, and then added to the parent rule (Alg. 3 line 13). Now the child rules support has to be calculated by calling the function *cardinalityQueries*. To reduce runtime, the positive example tables of the parent rules are saved in the Map *dfMap* and can be reused (Alg. 3 line 8) in the next iteration. If it is the first iteration of the algorithm and there is no table for the parent rule, which is of length one in the algorithm's first iteration, it is produced by filtering the facts table for the parent rules predicate (Alg. 3 line 4 to 7). This table will be called *instantiation table* as it contains all instantiations of the rule. The result of *cardinalityQueries* is unified with the result of previous iterations and is saved in the variable *whole* (Alg. 3 line 15 to 19), which is returned at the end of the function (Alg. 3 line 22).
cardinalityQueries, described in Algorithm 4, takes the table of the positive examples of the parent rule as *tpArDF* and the list of atoms of the child rule as parameters. To generate all positive examples the instantiated table of the newly added atom (Alg. 4 line 3), the last atom in the list, has to be joined with the

table of positive examples of the parent rule (Alg. 4 line 4). With the help of the *checkSQLWHERE* String this joined table can be filtered for correctly instantiated examples. In this String the positions of variables that are equal in the rule are put together, so that, when *checkSQLWHERE* is used, when passing queries in SQL SPARK, the same positions in the example tables are checked for equality. For the rule $isMarriedTo(a, c) \Rightarrow livesIn(a, b)$ the example table would have the first column as instantiations of the atom *livesIn(a,b)* and the second column as instantiations of *isMarriedTo(a,c)*, because the table is built from left to right beginning with the head. Keeping that in mind, in the String *checkSQLWHERE* the first column at the subject position would be compared to the subject position of the second column, as both positions contain the variable *a*. After the filtering (Alg. 4 line 7), the positive example table is saved in the *dfMap* (Alg. 4 line 8). Because every row has to be connected to a key, so that the positive examples can be successfully counted later on, the table is also joined with a key generated in line 6 (Alg. 4 line 9). This table is then returned (line 10).

Algorithm 3 Building of new Rule

```
1: function countProjectionQueriesDF(tpAr, variablesNewAtom)
2:     whole= null
3:     tpArDF = null
4:     if dfMap not initialized then
5:         table = this.dfTable
6:         tpArDF = "SELECT rdf AS tp0 FROM table WHERE rdf.predicate = '"
   + tpAr(0).predicate + "'"
7:     else
8:         tpArDF = get previously generated DataFrame with table of all positive
   examples for rule tpAr
9:     end if
10:    for all x ∈ relationSize do
11:        for all i ∈ variablesNewAtom do
12:            temp = tpAr.clone()
13:            temp += RDFTriple(i1, x1, i2)
14:            part = cardinalityQueries(tpArDF, temp)
15:            if whole == null then
16:                whole = part
17:            else
18:                whole = whole.unionAll(part)
19:            end if
20:        end for
21:    end for
22:    return whole
23: end function
```

Algorithm 4 Generation of all positive Examples

```
1: function CARDINALITYQUERIES(tpArDF, wholeAr)
2:     table = this.dfTable
3:     newColumn = "SELECT rdf AS tp" + (wholeAr.length - 1) + " FROM table
   WHERE rdf.predicate = '" + (wholeAr.last).predicate + "'"
4:     t = "SELECT * FROM tpArDF JOIN newColumn"
5:     checkSQLWHERE = String that links same variables together so that the
   DataFrame is filtered only for positive examples
6:     keyTable = DataFrame consisting of a String of the new atom and the position
   of the rule that is being refined in the iteration list
7:     lastTable = "SELECT * FROM t " + checkSQLWHERE
8:     setDfMap(calcName(wholeAr), last)
9:     out = "SELECT * FROM lastTable JOIN keyTable"
10:    return out
11: end function
```

3.1.2 Acceptance for Output To decide if a rule can be output, *AcceptedForOutput()* (Algorithm 7) checks the rule if it is closed, its *PCAConfidence* is greater than the *minConf* (Alg. 7 line 2) and the *PCAConfidence* of the rule is greater or equal the *PCAConfidence* of all parents of the rule (Alg. 7 line 8). To be able to check these thresholds, the *PCAConfidence* is needed. It is computed when a new rule is build and saved in the iteration list. In the following the process of computing the *PCAConfidence* will be described. The *PCAConfidence* can be written as the formula [3]:

$$pcaConf(\overrightarrow{B} \Rightarrow r(x,y)) = \frac{support(\overrightarrow{B} \Rightarrow r(x,y))}{PCABodySize(\overrightarrow{B} \Rightarrow r(x,y))}$$

The *PCABodySize* is the number of all negative and positive examples for a given rule. The function *cardPlusnegativeExamplesLength()* (Algorithm 5) computes and counts these examples. Because of the *Partial Completeness Assumption*, which says "if we know one y for a given x and r, then we know all y for that x and r", negative examples are found in connection with positive examples. Negative examples are bodies of a positive example, which are connected to the positive example only through the subject of the head of the positive example, but not the object. The bodies are positive examples for the body of the rule, but have no instantiation of the head to build a complete positive example.
cardPlusnegativeExamplesLength() uses this and counts all instantiations of the body of the rule that are connected to the subject of the head of a positive example. For this purpose, a table with all subjects of the heads of positive examples, the *subjectTable*, is computed (Alg. 5 line 3, PCA Calculation (2)). The facts table (cardTable, Alg. 5 line 2, PCA Calculation (1)) was computed in the previous iteration of the for loop. If the rule has 2 atoms, the table that will be counted can be build simply by filtering the facts table for the predicate of the rules body, the *bodyTable*, (Alg. 5 line 14) and then joining this table with *subjectTable* on the equality of the subjects in the *subjectTable* and the instantiations of the variable '?a' in the other table.

Algorithm 5 All negative and positive Examples

```
1: function cardPlusnegativeExamplesLength(tpAr)
2:     cardTable = DataFrame with all positive Examples for rule tpAr from dfMap
3:     subjects = "SELECT DISTINCT tp0.subject AS sub FROM cardTable"
4:     out = null
5:     if tpAr.length > 2 then
6:         out = negatveExampleBuilder(subjects, tpAr)
7:     else
8:         abString = ""
9:         if tpAr(1).subject == "?a" then
10:            abString = "subject"
11:        else
12:            abString = "`object`"
13:        end if
14:        twoLengthT = "SELECT rdf AS tp0 FROM dfTable WHERE
   rdf.predicate='" + (tpAr(1)).predicate + "'"
15:        Here the table with all facts that have the body's predicate as predicate is
   created
16:        out = "SELECT twoLengthT.tp0 FROM twoLengthT JOIN subjects ON
   twoLengthT.tp0." + abString + "=subjects.sub"
17:        Here the subject table is joined with the twoLengthT table on the position
   of variable ?a in the body. Variable ?a always represents the supject of the head.
18:    end if
19:    outCount = out.count()
20:    return outCount
21: end function
```

If the number of atoms of the rule is greater than 2 (Alg. 5 line 5) the *bodyTable* has to be build in a few steps before it can be joined with the *subjectTable*. This happens in the function *negativeExampleBuilder()* which is described in Algorithm 6. *negaitveExampleBuilder()* takes the list of atoms of the rule and the *subjectTable* as parameters and returns a DataFrame of all negative and positive examples. The building of the *bodyTable* is done by iterating over the list of atoms of the rules body and joining the result of the last iteration with the instantiation table of the current atom (Alg. 6 line 3 to 7, PCA Calculation (3)). For the *bodyTable* only to contain correctly linked atoms, all positions with the same variable (subject and object position) have to be checked for equality (Alg. 6 line 8, 10, PCA Calculation (4)). The process of building the example table and filtering for correct instantiations of the rule was also used when building the table with positive examples, but it was possible there to reuse previously build example tables instead of building the whole table from scratch. To be able to join the *subjectTable* and the *bodyTable*, the exact position of the variable '?a', which always has the subject position in the head, in one of the atoms is needed and is therefore saved in a Tuple named *abString* (Alg. 6 line 9, PCA Calculation (5)). In line 11 of Algorithm 6 both tables are, like in the example of length two, joined. The result of this join is returned. In Algorithm 5 the

bodyTable is counted and its result, the *PCABodySize*, returned, so it can be used as denominator in the *PCAConfidence* calculation.

3.1.3 PCAConfidence Calculation The rule that is used as an example is:
$livesIn(?c, ?b) \wedge isMarriedTo(?c, ?a) \Rightarrow livesIn(?a, ?b)$
(1)
The previously saved table with all positive examples for the rule is loaded.

livesIn	livesIn	isMarriedTo
(Sara,London)	(Steve,London)	(Steve,Sara)
(Linda,Berlin)	(John,Berlin)	(John,Linda)
(Sandra,Paris)	(Paul,Paris)	(Paul,Sandra)
(Lisa,Rome)	(James,Rome)	(James,Lisa)

Table 2: cardTable

(2)

The Subject table is created.

livesIn
Sara
Linda
Sandra
Lisa

Table 3: subjectTable

(3)
The rule has the length three, so *negativeExampleBuilder()* is used to build the *bodyTable*. For that you need the facts table:

tp0
livesIn(Sara,London)
livesIn(Steve,London)
livesIn(Linda,Berlin)
livesIn(John,Berlin)
livesIn(Sandra,Paris)
livesIn(Paul,Paris)
livesIn(Lisa,Rome)
livesIn(James,Rome)
livesIn(Peter,Paris)
livesIn(John,Paris)
livesIn(Mark,Moscow)
isMarriedTo(Steve,Sara)
isMarriedTo(John,Linda)
isMarriedTo(Paul,Sandra)
isMarriedTo(James,Lisa)
isMarriedTo(Peter,Sara)
isMarriedTo(John,Sara)
isMarriedTo(Mark,Mandy)
isCapitalOf(Berlin,Germany)
...

Table 4: facts table

The variable *complete* is initialized with the instantiation table for the first element of the body (Alg. 6 line 3). The body of the rule is: $livesIn(?c, ?b) \wedge isMarriedTo(?c, ?a)$.

tp0
livesIn(Sara,London)
livesIn(Steve,London)
livesIn(Linda,Berlin)
livesIn(John,Berlin)
livesIn(Sandra,Paris)
livesIn(Paul,Paris)
livesIn(Lisa,Rome)
livesIn(James,Rome)
livesIn(Peter,Paris)
livesIn(John,Paris)
livesIn(Mark,Moscow)

Table 5: All facts with the predicate *livesIn*

The new column, *newColumn*, that will be joined with the table *complete* is:

tp1
isMarriedTo(Steve,Sara)
isMarriedTo(John,Linda)
isMarriedTo(Paul,Sandra)
isMarriedTo(James,Lisa)
isMarriedTo(Peter,Sara)
isMarriedTo(John,Sara)
isMarriedTo(Mark,Mandy)

Table 6: All facts with the predicate *isMarriedTo*

The for loop (Alg. 6 line 4) takes the table *complete*, which contains the partial unfiltered body table that has been build through joins with new columns in previous iterations, and joins it with the instantiated table of the next element of the body. The following table shows the final result for the for loop of the example.

livesIn	isMarriedTo
(Steve,London)	(Steve,Sara)
(Steve,London)	(John,Linda)
(Steve,London)	(Paul,Sandra)
(Steve,London)	(James,Lisa)
(Steve,London)	(Peter,Sara)
(Steve,London)	(John,Sara)
(Steve,London)	(Mark,Mandy)
(John,Berlin)	(John,Linda)
(Paul,Paris)	(Paul,Sandra)
(James,Rome)	(James,Lisa)
(Peter,Paris)	(Peter,Sara)
(John,Paris)	(John,Sara)
...	...
(Sara,London)	(Steve,Sara)
(Sara,London)	(John,Linda)
(Sara,London)	(Paul,Sandra)
(Sara,London)	(James,Lisa)
(Sara,London)	(Peter,Sara)
(Sara,London)	(John,Sara)
(Sara,London)	(Mark,Mandy)
...	...

Table 7: *complete* and *newColumn* joined; This table is called *complete* and contains the cartesian product of the two tables

(4)
This table contains all combinations, not only correct instantiations of the rules body. For example, the second row of the table has no connection in the variables whatsoever. For this, *checkSQLWHERE* is build (Alg. 6 line 8). It makes it so, that every position in the rule that has the same variable is compared. Only the variable ?c appears more than once in the body, so only ?c is taken into consideration. For this example ?c is at the subject position in the first element of the body and at the subject position of the second element of the body. Therefore tp0.subject and tp1.subject of the table *complete* have to be equal. This is saved into the *checkSQLWHERE* String. Because of that, when the sql query is run in line 10 of Alg. 6, rows like the second row are discarded, as *Steve* and *John* are not equal. The resulting table is:

livesIn	isMarriedTo
(Mark,Moscow)	(Mark,Mandy)
(Steve,London)	(Steve,Sara)
(John,Berlin)	(John,Linda)
(Paul,Paris)	(Paul,Sandra)
(James,Rome)	(James,Lisa)
(Peter,Paris)	(Peter,Sara)
(John,Paris)	(John,Sara)

Table 8: positive and negative examples, unfiltered, containing invalid rows

(5)
The final table, that is output, is the *complete* table joined with the *subjectTable*. The resulting DataFrame can be counted, as the result is the *PCABodySize*. To join both tables on the subject of the head, the position and column of the variable '?a' in the body has to be determined. The variable of the subject position in the head is always '?a', as all heads are initialized with ('?a', predicate,'?b'). The position of of '?a' in the body is saved in the variable *abString* as a Tuple of (position of '?a' in the atom, position of the atom in the body) (Alg. 6 line 9). Therefore the *abString* in our example is the Tuple (tp1,object) as '?a' is in the object position in the second atom of the body, which is in the second column (tp1) of the *complete* table. So the first row of the table is discarded, because it does not contain a valid subject at this position. The following table is the final result:

livesIn	isMarriedTo
(Steve,London)	(Steve,Sara)
(John,Berlin)	(John,Linda)
(Paul,Paris)	(Paul,Sandra)
(James,Rome)	(James,Lisa)
(Peter,Paris)	(Peter,Sara)
(John,Paris)	(John,Sara)

Table 9: Positive and negative examples

livesIn	livesIn	isMarriedTo
(Sara,London)	(Steve,London)	(Steve,Sara)
(Linda,Berlin)	(John,Berlin)	(John,Linda)
(Sandra,Paris)	(Paul,Paris)	(Paul,Sandra)
(Lisa,Rome)	(James,Rome)	(James,Lisa)
-	(Peter,Paris)	(Peter,Sara)
-	(John,Paris)	(John,Sara)

Table 10: The positive and negative examples table with the head added back in, the positive examples have a head, the negative examples have none

Algorithm 6 DataFrame of negative and positive Examples

1: **function** NEGATVEEXAMPLEBUILDER(subjects, wholeAr)
2: wholeAr.remove(0)
3: complete = "SELECT rdf AS tp" + 0 + " FROM dfTable WHERE rdf.predicate = '" + (wholeAr(0)).predicate + "'"
4: **for** $i = 1$ to $wholeAr.length - 1$ **do**
5: newColumn = "SELECT rdf AS tp" + i + " FROM dfTable WHERE rdf.predicate = '" + (wholeAr(i)).predicate + "'"
6: complete = spark.sql("SELECT * FROM complete JOIN newColumn")
7: **end for**
8: checkSQLWHERE = String that links same variables together so that the DataFrame is filtered for positive and also negative examples because variables of the head are not taken into account
9: abString = Tuple(position of "?a" in atom, position of atom in wholeAr)
10: last = "SELECT * FROM complete " + checkSQLWHERE
11: out = "SELECT * FROM lastTable JOIN subjects ON last." + $abString_2$ + "." + $abString_1$ + "=subjects.sub"
12: **return** out
13: **end function**

Algorithm 7 When To Output

```
1: function acceptedForOutput(r, outMap, minConf)
2:     if ¬(r.closed()) ∨ r.getPcaConfidence() < minConf then
3:         return false
4:     end if
5:     parents = r.parentsOfRule(outMap)
6:     if r.getRule.length > 2 then
7:         for all rp ∈ parents do
8:             if r.getPcaConfidence() ≤ rp.getPcaConfidence() then
9:                 return false
10:            end if
11:        end for
12:    end if
13:    return true
14: end function
```

3.2 Different Approaches to shorten Runtime

In comparison to AMIE+, AMIESpark does not use an in memory database to compute the support, but computes the tables needed to calculate the support at runtime and saves the intermediate results for later use. In the process a huge amount of tables is produced by joining new columns to previously generated tables. The saving as well as the generation of these tables is expensive, so the time AMIESpark needs to finish is much longer than the time AMIE+ needs for small data inputs. Whether this is also true for very large inputs will be determined later in the thesis. One of the possibilities to improve runtime was to change the amount of data and time when that data is counted to calculate the support. It is possible to count every table for each atom addition separately or to union it all and calculate the support by using the *reduceByKey()* function after a whole iteration of the algorithm. Or, like described in the refinement process, to union all tables of one rules refinement, which has proven to be the fastest solution. Other approaches to shorten the runtime are explained in the following.

3.2.1 Parquet Approach As the saving and joining with tables with positive examples is the most expensive part of the algorithm, the first approach was to use parquet files as a saving method for the data frames, because parquet files provide additional optimizations to speed up queries. Parquet is a columnar storage file format. Columnar storage has the advantages that you have access to single columns, it is space efficient and that it limits IO operations [16]. Reading and writing to parquet files is supported by SPARK SQL. After using parquet files as means to save DataFrames, various other measures were taken to tune the applications runtime. One suggestion was to repartition the data before joining, because one problem could be that the number of tasks used can be too small, with the consequence that the available CPU is not used.

This can lead to slowing down due to pressure on the garbage collection [17]. In [17] the increase of partitions through repartitioning is suggested. Some sources suggest the exact opposite, to make sure the number of partitions is not too large. A trial with different values for repartition lead to longer runtimes when using high partition numbers like 400, and had no effect when ranging from 10 to 1 partitions, as described in Table 2. Another approach was to use broadcast variables to save some reused tables, which also lead to slower runtimes. *spark.sql.autoBroadcastJoinThreshold* is a value that can be manipulated when submitting the application and "configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join" [18]. To increase that that value to 8000000000 should have made the runtime faster, which was not the case (Table 2). Even after all these trials there were no significant improvements, which lead to the two other approaches described in the next two paragraphs.

Changes	Runtime
no changes made	14 min
repartition(2048)	out of memory error
repartition(1000)	running for 3h, stopped manually
repartition(400)	running for 1h, stopped manually
repartition(100)	running for 1h, stopped manually
repartition(2)	14 min
broadcast variable in *countProjectionQueriesDF()*	28 min
broadcast variable in *cardinalityQuerie()*	14 min
autoBroadcastJoinThreshold=8.000.000.000	14 min

Table 11: Contains changes made to the code and resulting runtime. The short example from Table 13 is used and the application is run locally.

3.2.2 Pre-generation Approach The Pre-generation Approach was devised in an attempt to create something similar to the in memory database used in AMIE+. In AMIE+ the in memory database contains all combinations of subject, objects and predicates. It is build before the main applications starts. This is what the Pre-generation Approach does as well. An independent application that has to be run before the main application builds all positive example tables for all rules produced by using the add dangling atom and add closing atom operator. The number of table that is produced and saved as parquet files is the maximum rule length. The tables contain all examples of size $\{1, 2, \ldots, maxLen - 1, maxLen\}$. This is done by the functions *countProjectionQueriesDF* and *cardinalityQueries*, whose results are unified and saved as parquet files. The *cardinalityQueries* function, which produces the example tables, is removed from the main application. At the position where it would be called normally, the parquet file for the current rules example table is read. Because

the most expensive computations, the joining and creation of many tables, was removed, the runtime of the main application should have improved, but it even got longer. The first guess as to what causes this was that the tables themselves were too long to read in sufficient time. The configuration *spark.memory.fraction* could be increased to give the application more space to store the tables [21]. This did not bring any improvement (Table 3), so the second version of the Pre-generation Approach was to produce tables containing the rules with their support out of the big tables. These tables were used instead of using *reduceByKey()* in the *refinement* function. These tables were significantly shorter, which should have had an impact on runtime, if the table length was the problem. But the runtime stayed the same, which indicated that IO time was the cause for longer runtime. Bad IO time for reading parquet files made a new approach without the use of parquet files necessary.

Changes	Runtime
no changes made	1h
spark.memory.fraction = 0.7	1h
spark.memory.fraction = 0.5	1h
spark.memory.fraction = 0.3	1h
spark.memory.fraction = 0.1	1h

Table 12: Tests run locally on the small example of Table 13.

3.2.3 Saved in Memory Approach The final approach simply saves the produced positive example tables in memory, instead of saving them as parquet files. The runtime achieved by this is still slower than the one achieved by AMIE+, but it does not use parquet files as storage. Since parquet file storage could possibly encounter I/O configuration issues in a distributed setting when working for other users, we decided not to use them in the following approaches. The Saved Memory Approach was therefore a way to increase usability. One thing that has to be noted is that when submitting the application, driver and executor memory have to be increased, so that no out of memory errors occur.

3.3 Runtime

The knowledge bases used to test the runtime of the application are the YAGO2 knowledge base, called yago2KB, and a 47K sample, which the AMIE+ team generated randomly from the YAGO2 KB, the yago2sample. The full "YAGO contains 120M facts about 2.6M entities" [22]. SPARK is optimized for big data, so one of our assumptions was that AMIESpark would perform better than AMIE+ for large KBs than smaller KBs. So yago2KB and yago2sample are used to test this, but because they are too big to do fast tests on our local one

node cluster, we created a smaller sample called smallTest described in Table 13. To make sure good rules are produced, it contains predicates that are logically connected. To have a general idea of the size of some tables for some rules, Table 13 also contains the number of facts with a distinct predicate.

predicate	count
produced	616
actedIn	4.919
directed	938
created	9.685
isMarriedTo	1.667

Table 13: smallTest with 17.825 facts with 5 distinct predicates

In Table 14 the runtime of all three samples is described.

	KB Size	**In Memory Approach**
smallTest	18K facts	14 min
yago2sample	75K facts	25 min
yago2KB	120M facts	7,5 h

Table 14: In Memory and Parquet Approach in comparison

To our disappointment the runtime of AMIESpark is always slower than that of the AMIE+ algorithm, but it is still faster than the state of the art algorithms WARMR and ALEPH. Their runtimes, when run on YAGO2, stem from the AMIE+ paper [3] and are compared to the runtime of AMIESpark in Table 15. WARMR and ALEPH are both Inductive Logical Programming (ILP) systems. While WARMR uses breadth-first search to find frequent patterns, which are the rules that are output, ALEPH has its own scoring functions to determine a rules quality to find the best rules to output [3].

In Memory Approach	**WARMR**	**ALEPH**
7,5 h	18 h	4.96s to $>$ 1 day

Table 15: Runtime of ALEPH, WARMER and the In Memory Approach in comparison.

4 Evaluation

In the following, the implementation choices and the decision to use SPARK to implement AMIE+ with SPARK will be evaluated. The runtime is an important measurement for that, as the objective of this thesis is to improve the runtime of AMIE+ for large datasets. The percentage of correct rules that are mined could be a measure too, but AMIESpark did not make any changes to improve the rule mining itself. Values for that can be found in [3]. So the focus is on the parts of AMIESpark that are the most expensive to compute.

4.1 Evaluation of implementation choices

The main design choice that differs from the implementation of AMIE+ is not using a pre generated database and at runtime generated DataFrame tables instead. The motivation behind that was to use features of SPARK to implement the algorithm of AMIE+. The initial approach was not to use DataFrames and join the new columns, but to use RDDs. The positive examples were build by using the function *cartesian()* to produce every combination of the positive example RDD, of the rule that is being refined, and the new atoms instantiation RDD. This lead to out of memory errors. This problem was solved by using DataFrames. In that regard it was a great improvement. The application is now able to finish and returns a correct output. But in comparison to AMIE+ AMIESpark did not manage to improve the runtime as hoped. Other fine tuning like the choice to compute the support after every refinement step for a rule, lead to small improvements regarding runtime. For further improvement, the approximation of the PCA Confidence, which is described in the paper of AMIE+ [3], is used. It approximates the PCA Confidence for rules where the head variables x, y are connected through a single chain of existentially quantified variables $z_1, \ldots, z_{n-1}$ as for these the computation of confidence is the most expensive [3]. All in all, after making an effort to find the best way to implement AMIE+ using the mechanics of SPARK, the runtime still is not sufficient.

4.2 Evaluation of SPARK as tool to implement this algorithm

Like described in the paragraph above, using DataFrames and its inbuilt properties of optimization and parallelization made it possible to run the extremely expensive operation of generating the positive example DataFrames and then computing the support. This operation is so costly, because every time a JOIN is used, SPARK hast to read all partitions to find and bring together all values to all keys that are being joined, which leads to a shuffle [19]. Shuffling also happens when the support of the rule candidates is calculated, triggered by the function *reduceByKey()*. The shuffling and the fact that this many tables are generated makes the operation expensive. The approach chosen to implement the algorithm is not able to use SPARK's strong points to their full potential. The level of parallelism is limited, as data that was computed in a previous

iteration is needed. Also there are not many occasions when SPARK's lazy evaluation in form of transformations can be used. For example when using *map()* instead of *for* loops at key points in the code, for instance the building of the initial refinement table, their use did not bring any improvements, or there were serialization errors when other functions were used inside the *map*. Serialization errors are errors that occur when a class is not implementing the interface *Serializable*, which is a condition for SPARK to be able to use this class in parallel operations [20]. So, my conclusion is that the approach to use and compute the DataFrames at runtime is not appropriate to implement AMIE+.

4.3 Conclusion

Using SPARK makes it possible to generate the support of the rules at runtime and is able to parallelize some parts of the the algorithm. But AMIESpark is still not able to compete with the runtime of AMIE+, as not all parts of the code can be parallelized. Also JOINs, which are used to generate every example table, are one of the most expensive operations in SPARK. And even after testing different approaches, applying changes to configurations to tune SPARK's performance and testing with varying cluster sizes, the runtime was not improved in a sufficient way. As mentioned in the Evaluation of SPARK as a tool to implement the algorithm, there are many other factors that made it difficult to improve the runtime. So as concluded earlier, the chosen approaches do not bring improvement when used with knowledge bases of a size AMIE+ has problems with.

5 Summary and Future Development

Being able to implement an algorithm with a relatively new tool like SPARK has its advantages and downsides. Finding the most promising data structures and applying them to the AMIE+ algorithm, trying to use SPARK's embedded optimizations to the fullest, lead to many changes and multiple approaches, because new features were added while programming. For example, in the beginning DataFrames were not added to SPARK yet, so all example 'tables' were RDDs filled with tuples or triples, which would later on be the rows of the DataFrame tables. To build these RDDs the Cartesian product was used to build all possible combinations of two fact RDDs, instead of using JOINs. As SPARK evolved, so did the approaches. DataFrames were used with SQL queries and parquet files to store them efficiently. But the parquet files lead to IO problems, which could not be resolved with fine tuning of configurations, even when separated from the main program in the Pre-generation approach. So the the final approach was devised.
One approach to improve the algorithm in the future could be to use the T-box of the RDF KB as well to find connections between atoms faster. The current application is build of java style code and could be improved by embracing the

advantages of scala and SPARK alike. As the runtime is still not ideal, one future task could be to make the program more parallel and using scala native structures like case classes to avoid Serializable errors. One approach could be to use case classes instead of the RuleContainer class, which contains the rule with all its important attributes like the support. Finally different structures for the list q could be tested as well as some different settings for SPARK itself, because the final approach does not use any special configurations, other than increasing driver and executor memory.

References

[1] RDF, `https://www.w3.org/RDF/`, 25.06.2016

[2] Luis Galárraga, Christina Teioudi, Katja Hose, Fabian M. Suchanek "AMIE: Association Rule Mining under Incomplete Evidence in Ontological Knowledge Bases." in *20th International World Wide Web Conference*, 2013.

[3] Luis Galárraga, Christina Teioudi, Katja Hose, Fabian M. Suchanek "Fast Rule Mining in Ontological Knowledge Bases with AMIE+." in *VLDB Journal*, 2015.

[4] P. Cimiano, A. Hotho, S. Staab. "Comparing Conceptual Divisive and Agglomerative Clustering for Learning Taxonomies from Text." in *ECAI*, 2004.

[5] Andrea Bellandi, Barbara Furletti, Valerio Grossi, Andrea Romei "Ontology-Driven Association Rule Extraction: A Case Study." in *CONTEXT*, 2007.

[6] Victoria Nebot, Rafael Berlanga "Mining Association Rules from Semantic Web Data." in *IEA/AIE*, 2010.

[7] Daniel Fleischhacker, Johanna VÃ¶lker, Heiner Stuckenschmidt "Mining RDF Data for Property Axioms" in *OTM*, 2012.

[8] Tao Jiang, Ah-Hwee Tan "Mining RDF Metadata for Generalized Association Rules: Knowledge Discovery in the Semantic Web Era" in *WWW '06*, 2006.

[9] Zhichun Wang, Juanzi Li "RDF2Rules: Learning Rules from RDF Knowledge Bases by Mining Frequent Predicate Cycles" in *CoRR, December 2015*, 2015.

[10] Yang Chen1, Daisy Zhe Wang1, Sean Goldberg "ScaLeKB: scalable learning and inference over large knowledge bases" in *The VLDB Journal (2016)*, 2016.

[11] spark.apache.org, `https://spark.apache.org/docs/latest/rdd-programming-guide.html`, 5.10.2017.

[12] Cloudera, `https://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_spark_apps.html`, 5.10.2017.

[13] Cloudera, `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/`, 5.10.2017.

[14] Databricks, `https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html`, 6.10.2017.

[15] Databricks, `https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html`, 6.10.2017.

[16] Tutorialpoint, `https://www.tutorialspoint.com/spark_sql/spark_sql_parquet_files.htm`, 13.10.2017.

[17] Cloudera, `https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/`, 13.10.2017.

[18] Source Code Spark, https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/internal/SQLConf.scala#L115, 13.10.2017.
[19] org.apache.spark, http://spark.apache.org/docs/2.1.1/programming-guide.html#overview, 17.10.2017.
[20] Oracle Docs, https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html, 17.10.2017.
[21] spark.apache.org, https://spark.apache.org/docs/latest/configuration.html#memory-management, 19.10.2017.
[22] max planck institute, https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/amie/, 15.02.2018.

Eidesstattliche ErklÃrung

Ich versichere hiermit, dass die Bachelorarbeit mit dem Titel 'Association Rule Mining of Linked Data Using Apache Spark' von mir selbst und ohne jede unerlaubte Hilfe angefertigt wurde, dass sie noch an keiner anderen Hochschule zur PrÃ¼fung vorgelegen hat und dass sie weder ganz noch in AuszÃ¼gen verÃ¶ffentlicht worden ist. Die Stellen der Arbeit â“ einschlieÃŸlich Tabellen, Karten, Abbildungen usw. â“, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich in jedem einzelnen Fall kenntlich gemacht.

Ort, Datum:

Unterschrift: