Mining Rules with Constants from Large Scale Knowledge Bases

Xuan Wang, Jingjing Zhang, Jinchuan Chen(✉), and Ju Fan

School of Information, Renmin University of China, Beijing, China
jcchen@ruc.edu.cn

Abstract. Rules or constraints can be used to clean a knowledge base, or find new facts which should have been included. Recently there are many efforts on automatically mining rules from large scale knowledge bases. However, these rules usually contain no constants. In practice, we often need some detailed rules, for example, rules restricted to a special country or a special profession. One major challenge of appending constants lies in that there are large amount of constants, each of which can generate a new rule. Moreover, we have to choose appropriate granularity in order to trade off between the applicability and precision (or support and confidence in traditional rule mining terminology). In this paper, we propose a Spark based solution to mine rules with constants, a taxonomy based approach to control the granularity, and several techniques to improve the efficiency. We also conduct extensive experiments to evaluate the efficiency and effectiveness of our solution with comparison with the state of the art works.

Keywords: Knowledge base · Rule mining · SPARK · Big data

1 Introduction

Recent years, large-scale RDF knowledge bases have been constructed [1], such as Freebase, YAGO and DBLP. These knowledge bases store structured information about real-world entities and the relationships between entities, which are formed as RDF triple $(subject, predicate, object)$. RDF is an imporant tool for representing conceptual models like UML. There are also many applications built based on RDF knowledge bases, such as automated customer service, structured search and semantic graph search etc.

However, these knowledge bases are far from complete, missing many facts and also containing some incorrect data [2,3]. Thus recently there are lots of research on mining rules or constraints in order to complete knowledge bases [4–8]. These rules can be in different forms like function dependencies [9], denial constraints [7], and first-order logic formulas [10]. The rationale is that it should be a common rule for the whole knowledge base if it is correct for most triples. For example, suppose we find that the nationality of one person is probably the same as his/her birthplace, we may conclude the following rule ϕ:

$$isCitizenOf(x, y) \leftarrow wasBornIn(x, y) \tag{1}$$

© Springer Nature Switzerland AG 2018
J. C. Trujillo et al. (Eds.): ER 2018, LNCS 11157, pp. 521–535, 2018.
https://doi.org/10.1007/978-3-030-00847-5_38

Applying this rule to the knowledge base, we may find some missing facts about the nationality information.

Since the scale of these knowledge bases are quite large[1], traditional rule mining approaches like FOIL [11] or ILP [12] have limited scalability and cannot be applied to large-scale knowledge bases. The work in [10] proposed a Spark-based rule-mining approach and illustrates nice scalability. However, [10] can only obtain rules without constants. This may miss many high-quality rules. For example, the rule ϕ in Eq. 1 is not good enough because in some countries nationality does not depend on birthplaces. But apparently we can have some qualified rules if we restrict y to some countries like USA. That is, we refine ϕ to ϕ':

$$isCitizenOf(x,'USA') \leftarrow wasBornIn(x,'USA') \quad (2)$$

The rule ϕ' covers fewer facts than ϕ, but is obviously more precise. In the tasks of knowledge completion, we need rules with high precision. Thus it is necessary to introduce constants into rules.

The works in [7,8] are able to obtain rules with constants. But the approach proposed in [8] is not designed for the distributed computing environment and is hard to scale out for large data sets. The work in [7] focuses on denial constraints only. The denial constraints can only identify that some combinations of facts are incorrect. They are useful for detecting errors but not able to infer new facts.

The target of this paper is to mine Horn-clause formed rules with constants. Note that allowing constants improve the complexity greatly. Each rule (without constants) may have thousands of extended sub-rules by appending different constants. Similar to [10], we propose a Spark-based approach to obtain scalability. We find that the performance is quite poor when directly apply the approach of [10] to deal with rules with constants. The reason lies in that it takes too much time for loading and exchanging the large data set in the Spark cluster when evaluating candidate rules. We try to solve this problem by designing a novel method, which reduces the chances of data loading as much as possible.

We further find that it is necessary to finely control the granularity of the rules. In the process of learning rules, we can split a rule ϕ into sub-rules $\phi_1, \cdots, \phi_k$. Each $\phi_i (i = 1, \cdots, k)$ covers a sub-set of facts that are covered by ϕ, while the confidence[2] of ϕ_i may increase. Note that when we append a condition $x =' c'$ to ϕ, we restrict x in the most strict way. According to our experiments, we can only find a few rules when directly restricting variables to constants, because these rules are too strict to cover enough number of facts.

With this observation, we propose a taxonomy based approach to trade off between the granularity and the quality. Instead of restricting a variable x to constants, we try to restrict x to some domains. The domains are hierarchical and form a taxonomy tree. Hence we can control the granularity by restricting x to an appropriate node (domain) in the tree.

[1] Yago3 has more than 10 million facts, and Freebase has about 2.4 billion facts.

[2] We will define this in Sect. 2.3.

The contributions of this paper are listed as follows.

- We propose a scalable solution for mining rules with constants from large-scale knowledge bases.
- We design a series of techniques for reducing the chances of data loading and improve the efficiency.
- We conduct extensive experiments to evaluate the efficiency and effectiveness of our approach. We also utilize our solution for completing a real data set and report some interesting findings.

The remaining parts of this paper are organized as follows. We first explain the learning model and propose the taxonomy based approach in Sect. 2. Then we propose our solution in Sect. 3 and report our experimental results in Sect. 4. Section 5 summarizes the related works and finally we conclude this paper in Sect. 6.

2 Learning Model

In this section, we will discuss the model utilized when mining rules from knowledge bases, including language bias and the quality metrics. We will also propose a taxonomy-based model to refine rules by appending constants.

2.1 Language Bias

There are many different types of rules or constraints, which can be applied to RDF knowledge bases. In this paper, we focus on Horn clauses, which is a special kind of logic formulas with at most one positive atom in the head.

Horn clauses. A *Horn clause* consists of a head and a body, where the head is a single atom and the body is a set of atoms.

$$H(\bar{X}) \leftarrow \overrightarrow{B}(\bar{X})$$

where $H(\bar{X})$ is an atomic formula, $\overrightarrow{B}(\bar{X})$ is a conjunction of atomic formulas, and $\bar{X}$ is a vector of variables.

As an example, the rule in Eq. 1 specifies that we can infer that x is a citizen of y if x was born in y. Here, both $isCitizenOf$ and $wasBornIn$ are predicates and x,y are variables. When learning rules from RDF knowledge bases, all RDF predicates will be adopted as predicates such as *livesIn, wasBornIn, isLocatedIn and human* etc. Some of them receive only one argument, meaning that the argument variable has a property, e.g. $human(x)$ specifies that x is a human. The others receive two arguments, specifying that the two argument variables have a special relationship, e.g. $wasBornIn(x, y)$.

We will also introduce a special predicate, *equals*, in order to append constants into rules. This predicate receives two arguments. Specifically, $equals(x,' c')$ means that a variable x equals to a constant $'c'$. In this paper, we mark constants by circling them with quotes.

Furthermore, we only consider *connected* and *closed* Horn clauses to ensure that the result rules do not contain irrelative atoms. Two atoms are *connected* if they share at least one variable. A rule is *connected* if every two atoms in the rule are connected directly or transitively. A rule is *closed* when every variable appears more than once.

We also limit the length of all rules to be less than or equal to three. As reported by [10], 90.3% rules with length longer than or equal to four can be reduced to shorter rules. Hence these long rules provide little knowledge than short ones.

Take all the circumstances of permutations into account, all the rules (without constants) satisfying the above constraints can be grouped into the following six types.

1. $p(x, y) \leftarrow q(x, y)$
2. $p(x, y) \leftarrow q(y, x)$
3. $p(x, y) \leftarrow q(z, x) \wedge r(z, y)$
4. $p(x, y) \leftarrow q(x, z) \wedge r(z, y)$
5. $p(x, y) \leftarrow q(z, x) \wedge r(y, z)$
6. $p(x, y) \leftarrow q(x, z) \wedge r(y, z)$

2.2 Refining Rules with Constants

One of the major tasks of this paper is to refine rules with constants. An intuitive way of introducing constants is to replace some variables in rules by constants or append an atom with the *equals* predicate. For example, $isCitizenOf(x, y) \leftarrow equals(y, 'USA'), wasBornIn(x, y)$. Note that this rule is logically the same as $isCitizenOf(x, 'USA') \leftarrow wasBornIn(x, 'USA')$. Thus we will not count in the atoms with the predicate *equals* when calculating the length of a rule.

Appending the *equals* predicate looks like a matter of course for introducing constants. However, we find that there are only a few qualified rules with the *equals* predicate. When the support and confidence[3] thresholds are set to 100 and 0.7 respectively, there are only 26 rules can be mined from knowledge base[4].

Note that, when appending an atom with the *equals* predicate into a rule ϕ, we exactly replace ϕ with a set of sub-rules, each of which is bound with a different constant. For example, the rule $isCitizenOf(x, y) \leftarrow wasBornIn(x, y)$ will generate more than two hundred thousand sub-rules if we append an atom $equals(y, 'c')$ because there are 281,036 possible choices for $'c'$ in the knowledge base. It seems that the rules with the *equals* predicate are too refined and we need to adjust the granularity.

We observe that an argument of every predicate is bound with a specific domain defined in the schema of a knowledge base. For example, the x in $wasBornIn(x, y)$ must be a *person*. Moreover, a domain has sub-domains and sub-domains may also have sub-domains, which finally builds a taxonomy tree.

[3] We will explain the two metrics soon.

[4] In this paper, without otherwise specified, the default knowledge base is Yago3.

For example, the type *person* can be classified into *slave*, *worker* and so on. Therefore, instead of restricting x to some constant like 'Donald_Trump', it looks more meaningful to restrict x to some domains, e.g. 'USA_People'.

With this observation, we propose another way of refining rules with constants. In order to refine a rule ϕ, we append an atom with predicate $rdftype(x,'c')$, where $'c'$ can be the domain of x as defined by the schema or any sub-domain of it.

According to our experiments, the taxonomy-based approach obtains many more meaningful rules than constants-based approach, i.e. appending atoms with $equals$. Another merit of the taxonomy-based approach lies in that it is easy to control the granularity. We can trade off between the accuracy and coverage by choosing appropriate node in the taxonomy tree.

In the following, if a rule contains atoms of *equals* or *rdftype*, we call it *a rule with constants*. Otherwise, we call it a rule without constants.

2.3 Quality Metrics

We adopt the quality metrics in [8], i.e. the support (denoted by *supp*) and confidence (denoted by $conf$).

The support of each rule is defined as the number of distinct subject and object pairs in the head atom of all instantiations satisfying the rules in the knowledge base.

$$supp(H(x,y) \leftarrow \overrightarrow{B}(\bar{X})) := \#(x,y) : \exists z_1, ..., z_m : \overrightarrow{B}(\bar{X}) \wedge H(x,y) \quad (3)$$

Here H(x,y) denotes the head atom and $\overrightarrow{B}(\bar{X})$ is the conjunction of a set of atoms in the body atom, and $z_1, ..., z_m$ are all the variables appearing in rule $\varphi(\bar{X})$ except x and y.

The confidence of each rule is defined to be the ratio of rule predictions in the knowledge base. It is obtained by dividing the support value by the number of distinct subject and object pairs in the head atom from all the instantiations satisfying the body atom in the knowledge base.

$$conf(H(x,y) \leftarrow \overrightarrow{B}(\bar{X})) := \frac{supp(H(x,y) \leftarrow \overrightarrow{B}(\bar{X}))}{\#(x,y) : \exists z_1, ..., z_m : \overrightarrow{B}(\bar{X})} \quad (4)$$

Lemma 1. ***Monotonity of Supp.*** *Suppose ϕ' is a rule refined from ϕ by appending an atom of $rdftype(x,'c')$ (x can be any variable appearing in ϕ), then $supp(\phi') \leq supp(\phi)$.*

Proof. Without loss of generalization, suppose ϕ is $H(x,y) \leftarrow \overrightarrow{B}(\bar{X})$, and ϕ' is $H(x,y) \leftarrow \overrightarrow{B}(\bar{X}) \wedge rdftype(x,'c')$. It is not hard to see that for every valuation $v' \vDash \phi'$, there must exist a valuation v which is a sub-set of v' and $v \vDash \phi$. Since the head variables $\{x,y\}$ are contained by both ϕ and ϕ', we can obtain that $v' \upharpoonright_{\{x,y\}} = v \upharpoonright_{\{x,y\}}$. Therefore, $\{v \upharpoonright_{\{x,y\}} | v \vDash \phi'\} \subseteq \{v \upharpoonright_{\{x,y\}} | v \vDash \phi\}$.

Here, $v \vDash \phi$ means that ϕ is satisfied by the assignment v, and $v \upharpoonright_{\overline{x}} = < a_1, \cdots, a_n >$, if $a_i = v(\overline{x}[i])$ $(i = 1, \cdots, n)$.

3 The Approaches

In this section, we will illustrate how to learn rules from knowledge bases. We will first explain how to learn rules based on the taxonomy tree in Sect. 3.1. Then we will analyze the bottleneck and propose two methods to improve the efficiency in Sects. 3.2 and 3.3.

3.1 Mining and Refining (MR)

Our first approach is called *mining and refining* (MR in short). As implied by the name, it contains two phases. In the first phase, we learn rules without constants from the knowledge base. Next, we try to refine the rules by appending the *rdftype* predicate.

The mining phase adopts the algorithm in [10]. The refining phase tries to extend each candidate rule ϕ to a set of sub-rules with constants. For each variable x in ϕ, we first load the domain c of x from the KB's schema. Then we traverse the taxonomy tree rooted at c. The traverse process is listed in Algorithm 1. It is basically a DFS. The lines 2–5 in Algorithm 1 specify the conditions of splitting a domain. We split a domain c if and only if the support score of the current rule (bound with domain c) is high enough and the confidence score is not qualified. Note that by splitting a domain, we may obtain rules with higher confidence scores, but the support scores cannot be improved (Lemma 1).

Algorithm 1. traverse(ϕ, c)

```
1 φ' ← rdftype(x,'c') ∧ φ ;
2 if supp(φ') < α then
3 |  Return;
4 else if conf(φ') > β then
5 |  Return;
6 else
7 |  for each sub-domain s of c do
8 |  |  traverse(φ, s);
```

In the experiments, we find that MR is quite slow. The reasons are two folds. Firstly, MR needs to extend each candidate rule ϕ to a set of sub-rules with constants by DFS, and for each sub-rule, we need to load the dataset to calculate its support and confidence score, which is very time-consuming. Secondly, each rule obtained in the mining phase has many candidate sub-rules, each of which requires an independent Spark job containing many unnecessary tasks. Next we will discuss how to improve the efficiency of MR.

Algorithm 2. MR-Batch

require : $facts1 = \{(pred, sub, obj)\}$,$facts2 = \{(pred, sub, obj)\}$
require : $rules = \{(ID, head, body)\}$,$types = \{(entity, type)\}$
1 **Join** facts1, facts2 and types on facts1.pred=rules.head and facts2.pred=rules.body and facts1.sub=facts2.obj and facts1.obj=facts2.sub and facts1.sub=types.entity;
2 **Distinct** the $\{(rules.head, rules.body, facts1.sub, facts1.obj, types.type)\}$;
3 **GroupBy** types.type, yielding a list of $\{(rules.head, rules.body, facts1.sub, facts1.obj)\}$ pairs for each type;
4 **Count** $\{(rules.head, rules.body, facts1.sub, facts1.obj)\}$ pairs for each type, yielding a list of $\{(type, supp)\}$ pairs;
5 **Join** supp and denominator(the denominator is calculated similar to the way of calculating supp), yielding a list of $\{(type, supp, deno)\}$;
6 **Map** $\{(type, supp, deno)\}$ to {(type, supp, supp/deno)};

3.2 MR-Batch

The MR algorithm needs to evaluate the quality of each refined rule, which is very inefficient because there are millions of refined rules and each of which requires an independent Spark job for quality evaluation. In this section, we propose an approach named MR-Batch, which evaluates the quality scores of all rules refined from one rule in the same Spark job.

Like MR, the MR-Batch approach also contains two phases, mining and refining. The difference lies in that now we replace Algorithm 1 with Algorithm 2. In Algorithm 1, we need to invoke a Spark job each time when we evaluate a rule. However, in Algorithm 2, we invoke only one Spark job if we want to refine a rule ϕ.

As illustrated in Algorithm 2[5], in Step 1 and Step 2, we join the tables and use the Distinct operator to deduplicate (x, y) pairs. Then, we group all tuples with the same key ($types.type$) into one set and count $\{(rules.head, rules.body, facts1.sub, facts1.obj)\}$ pairs for each type, yielding a list of $\{(type, supp)\}$ pairs. The next step is to calculate the denominator of confidence and join the support and denominator scores. Finally, map each $\{(type, supp, deno)\}$ to $\{(type, supp, supp/deno)\}$ to yield a list of $\{(type, supp, conf)\}$.

3.3 Loading Data only once (LDOO)

MR-Batch is much more efficient than MR. Actually, for the tasks of mining rules from Yago3, MR cannot finish in two days, while the MR-Batch takes only about two hours. But MR-Batch still wastes too much time on repeatedly loading data and can be further improved.

[5] For ease of illustration, both Algorithms 2 and 3 are tailored for the second rule type.

We design a new algorithm named LDOO (Loading Data Only Once). The algorithm loads the whole data set and mines all rules through a single Spark task.

The basic idea of LDOO is simple. For a given rule type, like $p(x, y) \leftarrow q(y, x)$, we join facts (i.e. triples) according to the join conditions specified in the rule type. We also make another join with the *rdftype* triples with the join condition x or y. Then we divide the whole join results into groups by the predicates and the constants in *rdftype*. We utilize the taxonomy information to generate the *rdftype* triples. For example, if $rdftype(x,'worker')$, we will also have $rdftype(x,'person'$ since '*worker*' is a sub-type of '*person*'.

Algorithm 3. LDOO

require : $facts1 = \{(pred, sub, obj)\}$, $facts2 = \{(pred, sub, obj)\}$
require : $types = \{(entity, type)\}$
1. **Join** facts1, facts2 and types on facts1.sub=facts2.obj and facts1.obj=facts2.sub and facts1.sub=types.entity;
2. **Distinct** the {(facts1.pred, facts2.pred, facts1.sub, facts1.obj, types.type)};
3. **GroupBy** (facts1.pred, facts2.pred, types.type), yielding a list of {(facts1.sub, facts1.obj)} pairs for each key;
4. **Count** {(facts1.sub, facts1.obj)} pairs for each key, yielding a list of {((p, q, t), supp)} pairs;
5. **Join** supp and denominator(the denominator is calculated similar to the way of calculating supp), yielding a list of $\{((p, q, t), supp, deno)\}$;
6. **Map** $\{((p, q, t), supp, deno)\}$ to {((p, q, r, t), supp, supp/deno)};

The details of the LDOO algorithm are illustrated in Algorithm 3 with Spark primitives. In Step 1 and Step 2, we join tables and deduplicate the pairs. In Step 3 and Step 4, we group tuples with same key $\{(facts1.pred, facts2.pred, types.type)\}$ into one set and count the number of pairs for each key, yielding a list of $\{((p, q, t), supp)\}$ pairs. Then, calculate the denominator of confidence. Finally, we join support and denominator scores and map each $\{((p, q, t), supp, deno)\}$ to $\{((p, q, r, t), supp, supp/deno)\}$ to yield a list of $\{((p, q, t), supp, conf)\}$.

Note that the MR and MR-Batch algorithm are based on the taxonomy structure and can only mine rules with the *rdftype* predicate. But LDOO can learn rules with *rdftype* or *equals*. To do this, we just need to remove the join conditions with *rdftype* and add a GroupBy parameter x or y. In our experiments, we implement LDOO for both *rdftype* and *equals*.

Aggregation Pushdown. LDOO worked well for types 1 to 5, but it failed in mining rules for the six-th rule type. By checking the logs, we found that some workers crashed due to huge intermediate results. The six-th rule type, i.e. $p(x, y) \leftarrow q(x, z), r(y, z)$, requires a join on the *object* attribute as indicated by the variable z. However, many object values have high appearance frequency which results in large number of join results. In Yago, there are 10,502 objects which appear more than 100 times, 25 objects more than 10,000 times, and 2 objects (*i.e.'female' and 'male'*) even more than 100,000 times.

Based on this observation, we try to improve the performance of LDOO by pushing down the aggregation operator. The idea is to perform the COUNT operator before JOIN, in order to reduce the amount of intermediate results.

Figure 1 illustrates the original query plan of the mining task for the six-th rule type, and Fig. 2 shows the rewritten plan by pushing down the COUNT operators. According to [19,20], the two plans are equivalent.

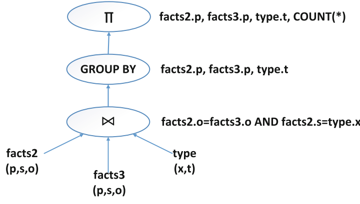

Fig. 1. Original plan

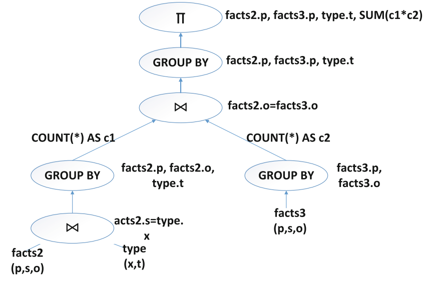

Fig. 2. Plan after pushing down COUNT

4 Experimental Study

In this section, we first summarize the settings of the experiments in Sect. 4.1. Then we report the efficiency and effectiveness results in Sects. 4.2 and 4.3 respectively.

4.1 Settings

We conduct all experiments on a 18-nodes cluster, consisting of one master and 17 workers. All these nodes are hosted on a cloud platform. Each node is equipped with one 2.4 GHz processor and 500G hard disk space. The master has 8G RAM and while each worker node has 4G RAM. All the approaches are implemented in scala 2.11.8 and java 1.8.0. The jobs run on the Spark 2.1.0 platform, and the OS is CentOS 6.5.

All the experiments are conducted on two real datasets, YAGO 3.0.0[6] and DBPedia[7]. The statistics are listed in the following (Table 1).

Table 1. Statistics of the datasets

	Yago	DBPedia
# of Triples	10,400,678	52,680,098
# of Entities	4,464,016	14,592,204
# of Predicates	125	59,149
# of Types	569,312	385

[6] https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/.

[7] http://wiki.dbpedia.org/develop/datasets.

Compared Methods. We implement the three methods proposed in Sect. 3, i.e. MR, MR-Batch and LDOO. We also compare our methods with the one in [10]. Note that we also implement LDOO for mining rules with the *equals* predicate. However there are only a few rules can have enough *supp* scores, because the granularity of these rules are too small. When $conf$ is set as 0.6, we can obtain only 26 rules. Therefore, we only report the results of mining rules with the *rdftype* predicate in the following. There are two parameters in the experiments, i.e. confidence and support, whose default values are set as 0.6 and 100 respectively.

4.2 Efficiency Evaluation

First of all, we report the results of efficiency evaluation. The task is to mine rules with constants from the knowledge base. As illustrated in Fig. 3, the x-axis is the rule type (Sect. 2.1). Note that the y-axis is logarithmic. MR is the slowest one. Actually MR cannot finish in ten hours for each rule type and we have to terminate it. MR-Batch is faster than MR, which requires about five hours. LDOO is the best, which costs about thirty minutes. As explained in Sect. 3.3, LDOO saves the expensive cost of reloading data from disk and moving data among the network. There is an exceptional case in Rule-Type 5, where MR-Batch beats LDOO. The reason is there are only a few (less than 10) rules to be refined for this type, which greatly reduces the workload of MR-Batch. Figure 4 illustrates the mining time on DBPedia. Note that we only report the result of LDOO here because neither MR nor MR-Batch can finish the mining job on DBPedia in 24 h.

Next we take a breakdown analysis for MR-Batch and show the results in Fig. 5. As discussed in Sect. 3.2, MR-Batch contains two phases, mining and refining. As illustrated in Fig. 5, the refining phase dominates the time cost. The reason is that we need to invoke a SPARK job for each rule obtained in the mining phase in order to evaluate the candidate sub-rules.

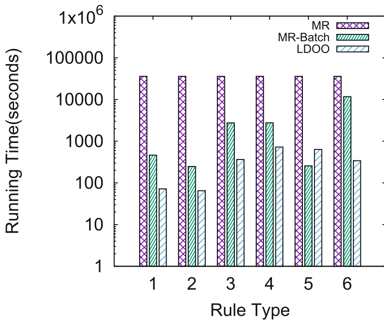

Fig. 3. Rule mining time (Yago)

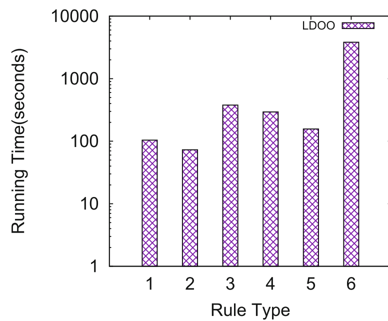

Fig. 4. Rule mining time (DBPedia)

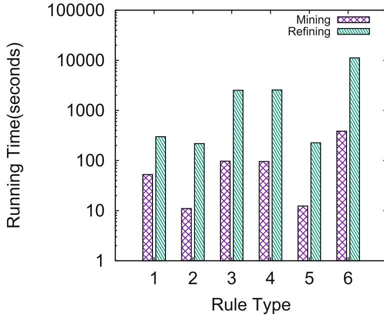

Fig. 5. Breakdown of MR-Batch (Yago)

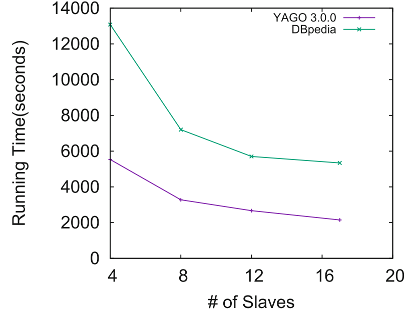

Fig. 6. Scalability (Yago & DBPedia)

We also conduct an experiment to evaluate the scalability of LDOO. As shown in Fig. 6, when the number of worker nodes increases, the time cost of LDOO decreases. For example, it takes 5,532 s when there are only 4 nodes and 2,149 s for 17 nodes, when mining rules from Yago.

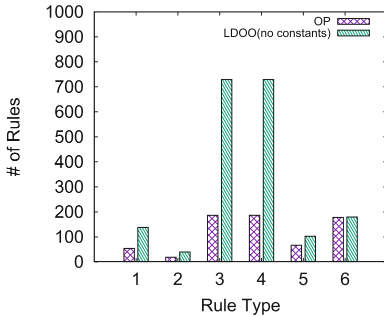

Fig. 7. Comparison of # of rules (Yago)

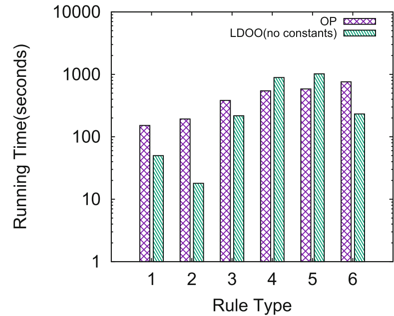

Fig. 8. Comparison of time (Yago)

Comparison with Ontological Pathfinding (OP). The OP approach in [10] depends on the domains of predicates when generating candidate rules. For example, when generating a rule $p(x, y) \leftarrow q(x, y)$, OP requires that the subject/object domains of predicates p, q must overlap because they share the same variables x, y. However, the schema data of Yago are not complete. Therefore, OP misses many meaningful rules.

As illustrated in Fig. 7, LDOO obtains several times more rules than OP, because LDOO does not have this limit of domain overlapping. Moreover, the time cost of LDOO is similar to OP (Fig. 8). Note that here we restrict LDOO to find rules without constants in order to make a fair comparison. Hence in the figure, the label is LDOO(no constants).

4.3 Effectiveness Evaluation

Number of Rules. Firstly, we conduct an experiment about how the number of rules varies with the confidence parameter. The results are shown in Fig. 9. We can find that LDOO always outperforms MR-Batch. Therefore, LDOO always obtains more rules than MR-Batch in spite of the confidence threshold.

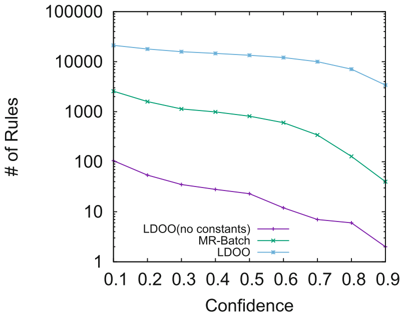

Fig. 9. # of rules (Yago)

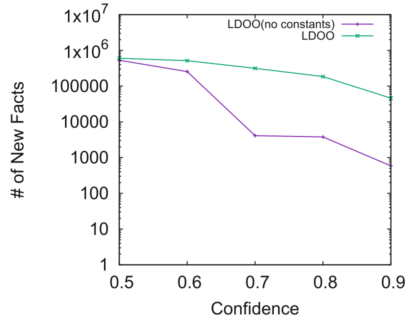

Fig. 10. # of new facts (Yago)

Knowledge Base Completion. In this task, we apply the mined rules to automatically replenish knowledge base, which is basically to infer new facts based on the qualified rules. For example, suppose we have the following rule:

$$hasChild(x, y) \leftarrow isMarriedTo(x, z), hasChild(z, y), \\ rdftype(x, Russian_grand_dukes)$$

The rule above means: suppose we have three triples, $isMarriedTo(x, z)$, $hasChild(z, y)$ and $rdftype(x, Russian_grand_dukes)$, we can then infer that the triple $hasChild(x, y)$ must be correct and put it into the knowledge base if it is missing.

First of all, we did several experiments to find an appropriate value for the $conf$ parameter. We randomly select 50 new facts for each $conf$ setting and manually check the correctness. The results are $0.92, 0.92, 0.96, 1.0$ for $conf = 0.5, 0.6, 0.7, 0.8$ respectively.

Figure 10 illustrates how the number of new facts varies with the confidence parameter. We can find that LDOO outperforms LDOO(no constants). When the confidence is set as 0.7, we can obtain more than 200,000 new facts by the rules with constants (about 96% of these new facts are correct). However, if there are no constants in the rules, we can only obtain about 3,000 new facts for the same confidence. This result confirms the effectiveness of our solution for completing knowledge bases.

Table 2 shows several example rules mined by LDOO. The third column of Table 2 compares the confidence values of the rule and its parent rule (i.e. the corresponding rule without constants). Clearly the refined rules have higher confidence values than their parent rules. We also list several inferred new facts in Table 3.

Table 2. Example rules

Rule	rdftype of x	Conf.
$playsFor(x, y) \leftarrow isAffiliatedTo(x, y)$	$football_player$	0.78(0.59)
$isLocatedIn(x, y) \leftarrow wasBornIn(z, x), isCitizenOf(z, y)$	$Cities_in_California$	0.87(0.56)
$hasFamilyName(x, y) \leftarrow hasChild(z, x), hasFamilyName(z, y)$	$lawyer$	0.72(0.51)

Table 3. New facts inferred by rules

New facts
playsFor(Oleksiy_Antonov,Ukraine_national_football_team)
hasFamilyName(Sam_Houston,“Kaufman”@eng)
isLocatedIn(Moscow,Russian_Federation)

We put the whole set of rules and inferred new facts in a public web site. Interested readers can find them at the following url:

https://pan.baidu.com/s/1oAqprqa (pass: 8m8d)

5 Related Works

Rule learning is a classical AI task which aims at learning first-order logic formulas from given data sets [13]. Classical rule learning methods [11,12,14] rely on some traverse strategies to search and evaluate all candidate rules, which are hard to deal with large data sets. Some recent works [8,15] improve the efficiency by running mining tasks in parallel, but it is not a distributed method and hard to scale out.

Chen et al. [4,10] proposed an ontology-based routing algorithm OP [16]. The whole task is divided into many sub-tasks, which run in parallel on a Spark platform. The limitation of OP lies in that it excludes constants and misses many valuable rules.

The works [7,17] admit constants, but both have limitations. [7]'s language model is quite limited and not able to find new facts which should be included in knowledge bases, while [17]'s implementation is for single machine and hard to be applied in large data sets.

In this paper, we design and implement our approaches based on the Spark platform [18]. Spark can provide an efficient and general data processing platform. Its core consists of a set of powerful and high-level libraries for large-scale parallel and distributed data processing.

6 Conclusion

Mining rules with constants is important for knowledge base construction. This paper proposes a scalable solution based on the Spark platform, including a series of approaches to improve both the efficiency and quality. In the future work, we will study how to generalize the learning model, e.g. introducing negative atoms. We will also study how to further improve the efficiency by optimizing the mining process.

Acknowledgments. This work was supported by the National Key Research & Develop Plan (No. 2016YFB1000702), National Science Foundation of China (No. 61602488), and Talent Training Fund at RUC.

References

1. Mahdisoltani, F., Biega, J., Suchanek, F.M.: YAGO3: a knowledge base from multilingual wikipedias. In: Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, 4–7 January 2015, Online Proceedings (2015)
2. Niu, F., Zhang, C., Ré, C., Shavlik, J.W.: DeepDive: web-scale knowledge-base construction using statistical learning and inference. In: Proceedings of the Second International Workshop on Searching and Integrating New Web Data Sources, Istanbul, Turkey, 31 August 2012, pp. 25–28 (2012)
3. Shin, J., Wu, S., Wang, F., De Sa, C., Zhang, C., Ré, C.: Incremental knowledge base construction using DeepDive. PVLDB **8**, 1310–1321 (2015)
4. Chen, Y., Wang, D.Z., Goldberg, S.: ScaLeKB: scalable learning and inference over large knowledge bases. VLDB J. **25**, 893–918 (2016)
5. Lao, N., Mitchell, T., Cohen, W.W.: Random walk inference and learning in a large scale knowledge base. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, pp. 529–539 (2011)
6. Chen, Y., Wang, D.Z.: Knowledge expansion over probabilistic knowledge bases. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 649–660 (2014)
7. Chu, X., Ilyas, I.F., Papotti, P.: Discovering denial constraints. Proc. VLDB Endow. **6**, 1498–1509 (2013)
8. Galárraga, L.A., Teflioudi, C., Hose, K., Suchanek, F.: 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 (2013)
9. Fan, W., Geerts, F., Li, J., Xiong, M.: Discovering conditional functional dependencies. IEEE Trans. Knowl. Data Eng. **23**, 683–698 (2011)
10. Chen, Y., Goldberg, S., Wang, D.Z., Johri, S.S.: Ontological pathfinding: mining first-order knowledge from large knowledge bases. In: Proceedings of the 2016 International Conference on Management of Data, pp. 835–846 (2016)
11. Quinlan, J.R.: Learning logical definitions from relations. Mach. Learn. **5**, 239–266 (1990)
12. Muggleton, S., De Raedt, L.: Inductive logic programming: theory and methods. J. Log. Program. **19**, 629–679 (1994)
13. Frnkranz, J., Gamberger, D., Lavrac, N.: Foundations of Rule Learning. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-540-75197-7

14. Savasere, A., Omiecinski, E., Navathe, S.B.: An efficient algorithm for mining association rules in large databases. In: Proceedings of the 21st International Conference on Very Large Data Bases, pp. 432–444 (1995)
15. Galárraga, L., Teflioudi, C., Hose, K., Suchanek, F.M.: Fast rule mining in ontological knowledge bases with AMIE$$+$$+. VLDB J. **24**, 707–730 (2015)
16. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: VLDB 1994, Proceedings of 20th International Conference on Very Large Data Bases, 12–15 September 1994, Santiago de Chile, Chile, pp. 487–499 (1994)
17. Zeng, Q., Patel, J.M., Page, D.: QuickFOIL: scalable inductive logic programming. Proc. VLDB Endow. **8**, 197–208 (2014)
18. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
19. Yan, W.P., Larson, P.: Eager aggregation and lazy aggregation. VLDB **31**(12), 345–357 (1995)
20. Harinarayan, V., Gupta, A.: Generalized projections: a powerful query-optimization technique (1995)