Ontological Pathfinding: Mining First-Order Knowledge from Large Knowledge Bases

Yang Chen† Sean Goldberg† Daisy Zhe Wang† Soumitra Siddharth Johri‡
Department of Computer and Information Science and Engineering
University of Florida
†{yang,sean,daisyw}@cise.ufl.edu ‡soumitra.johri@ufl.edu

ABSTRACT

Recent years have seen a drastic rise in the construction of web-scale knowledge bases (e.g., Freebase, YAGO, DBPedia). These knowledge bases store structured information about real-world people, places, organizations, etc. However, due to limitations of human knowledge and information extraction algorithms, these knowledge bases are still far from complete. In this paper, we study the problem of mining first-order inference rules to facilitate knowledge expansion. We propose the Ontological Pathfinding algorithm (OP) that scales to web-scale knowledge bases via a series of parallelization and optimization techniques: a relational knowledge base model to apply inference rules in batches, a new rule mining algorithm that parallelizes the join queries, a novel partitioning algorithm to break the mining tasks into smaller independent sub-tasks, and a pruning strategy to eliminate unsound and resource-consuming rules before applying them. Combining these techniques, we develop the first rule mining system that scales to Freebase, the largest public knowledge base with 112 million entities and 388 million facts. We mine 36,625 inference rules in 34 hours; no existing approach achieves this scale.

Keywords

Knowledge bases; data mining; first-order logic; scalability

1. INTRODUCTION

Recent years have seen tremendous research and engineering efforts in constructing large knowledge bases (KBs). Examples of these knowledge bases include DBPedia [4], DeepDive [39], Freebase [8], Google Knowledge Graph [7], NELL [9], OpenIE [5, 18], ProBase [53], and YAGO [6, 24, 35]. These knowledge bases store structured information about real-world people, places, organizations, etc. They are constructed by human crafting (DBPedia, Freebase), information extraction (DeepDive, OpenIE, ProBase), reasoning and inference [12, 46], knowledge fusion [52, 16], or combinations of them (NELL).

In this paper, we focus on the problem of mining first-order inference rules to support knowledge expansion. An inference rule is a first-order Horn clause to discover implicit facts. As an example, the following rule expands knowledge of people's birth places:

$$\text{wasBornIn}(x, y), \text{isLocatedIn}(y, z) \rightarrow \text{wasBornIn}(x, z).$$

Rules are useful in a wide range of applications: knowledge reasoning and expansion [12, 46], knowledge base construction [9], question answering [14], knowledge cleaning [19, 44], knowledge base maintenance [48], Markov logic learning [27], etc.

Mining Horn clauses has been studied extensively in inductive logic programming. However, today's knowledge bases pose several new challenges. First, knowledge bases are often prohibitively large. For example, as of this writing, Freebase has 112 million entities and 388 million facts. None of the existing rule mining algorithms efficiently support KBs of this size. Second, knowledge bases implement the open world assumption, implying that we have only positive examples for rule mining. To address these challenges, a number of new approaches are proposed: SHERLOCK [47], AMIE [20], Markov logic structure learning [26, 27], etc. Still, new techniques need to be invented to scale up state-of-the-art approaches to knowledge bases of billions of facts.

In this paper, we propose the Ontological Pathfinding algorithm (OP) to tackle the large-scale rule mining problem. We focus on scalability and design a series of parallelization and optimization techniques to achieve web scale. Following the relational knowledge base model [12], we store inference rules in relational tables and use join queries to apply them in batches. The relational approach outperforms state-of-the-art algorithms by orders of magnitude on medium-sized knowledge bases [12]. To scale to larger knowledge bases, we parallelize the mining algorithm by dividing the input knowledge base into smaller groups running parallel in-memory joins. The parallel mining algorithm can be implemented on state-of-the-art cluster computing frameworks to achieve maximum utilization of available computation resource.

Furthermore, even if we parallelize the mining algorithm, the parallel tasks are dependent on each other. In particular, the tasks need to shuffle data between stages. As the knowledge bases expand in scale, shuffling becomes the bottleneck of the computation. This shuffling bottleneck motivates us to introduce another layer of partitioning on top of the parallel computation, a partitioning scheme that divides the mining task into smaller *independent* sub-tasks. Each partition still runs the same parallel mining algorithm as before, but on a smaller input. Since each partition is independent from each other, the results are unioned in the end; no data exchange occurs during computation. Our experiments show that we accomplish the Freebase mining task within 34 hours that does not finish in 5 days without partitioning.

One major performance bottleneck is caused by large degrees of join variables in the inference rules. Applying these rules in the mining process generates large intermediate results, enumerating

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882954

all possible pair-wise relationships of the joined instances. As a result, these rules are often of low quality. Based on this observation, we use non-functionality as an empirical indication of inefficiency and inaccuracy. In our experiments, we determine a reasonable functional constraint and show that 99% of the rules violating this constraint turn out to be false. Removing those rules reduces runtime by more than 5 hours for a single mining task. Combining our approaches, we develop the first rule mining system that scales to Freebase, the largest public knowledge base with 112 million entities and 388 million facts. We mine 36,625 inference rules in 34 hours; no existing approach achieves this scale.

In summary, we make the following contributions to achieve scalability in mining first-order rules from large knowledge bases:

- We design an efficient rule mining algorithm that evaluates inference rules in batches using join queries. The algorithm runs in parallel by dividing the knowledge base into smaller groups running parallel in-memory joins;
- We design a novel partitioning algorithm to divide the mining task into small independent sub-tasks. Each sub-task is parallelized, running the same mining algorithm, but operates on small inputs. Data shuffling is unnecessary among sub-tasks;
- We define the non-functionality score to prune dubious and inefficient rules. We experimentally determine a reasonable functional constraint and show the constraint improves efficiency and accuracy of the mining algorithm;
- We conduct a comprehensive experiment study to evaluate our approaches over public knowledge bases, including YAGO and Freebase. Our research leads to a first rule set for Freebase, the largest knowledge base with hundreds of millions of facts. We publish the code and data repository online[1].

The remainder of this paper is organized as follows. Section 2 discusses related works. Section 3 defines the first-order mining problem. Section 4 introduces the underlying background knowledge. Section 5 introduces the Ontological Pathfinding algorithm in detail. Section 6 describes the partitioning algorithm. Section 7 presents the experiment study, and Section 8 concludes the paper.

2. RELATED WORK

Mining Horn clauses. Mining Horn clauses [25] has first been studied in Inductive Logic Programming (ILP) literature [41, 37, 42]. Recently, SHERLOCK [47] and AMIE [20] have extended it to mining first-order inference rules from knowledge bases by defining new metrics to address the open world assumption made by the knowledge bases. AMIE achieves the state-of-the-art efficiency using an in-memory database to support the projection queries for counting. Sherlock and AMIE have mined 30,912 and 1090 inference rules from 250K (OPENIE [5, 18]) and 948K (YAGO2 [24]) distinct facts, respectively. Still, none of these approaches scales to the Freebase size. To solve this scalability problem, we adopt the mining model of AMIE, described in Section 3, and scale it up using a series of parallelization and optimization techniques.

Parallel computing. In recent years, various data processing frameworks have been proposed and developed to facilitate large-scale data analytics, including in-databases analytics [23, 51, 31, 38, 50], MapReduce [15], FlumeJava [11], Spark [55, 54], GraphLab [34, 33], Datapath [13, 3], etc. These systems are effective in a variety of data mining and machine learning problems. Given no previous work that scales up the rule mining algorithms to the web, we are motivated to leverage state-of-the-art parallel computing techniques. The parallel mining algorithm we propose combines the relational model [12] and the MapReduce programming paradigm [15], consisting of a sequence of parallel operations on the KB tables. We implement and evaluate it on Spark in favor of its efficient pipelining of parallel operations using distributed caches.

Functional Constraints. Constraints have proven helpful in a wide range of knowledge base construction and expansion problems. NELL employs a set of coupling constraints to prevent "semantic drifts" in constructing the knowledge base [10]. A particularly useful class of constraints is functional constraints. The N-FOIL algorithm [30] for NELL uses functional constraints to provide negative examples for ILP learners. [32, 44] mines functional constraints from automatically constructed knowledge bases. [12] applies functional constraints to detect contradictions and ambiguous entities for the knowledge expansion problem. [46] leverages functionality properties of predicates to scale up textual inference to the web. These works of speeding up knowledge and textual inference tasks by leveraging functionality properties of predicates inspire us to apply them to the mining problem by extending the notion of functionality to Horn clauses to prune erroneous rules.

Mining association rules. Since the first introduction of association rule mining in [1], researchers have developed a number of improvements [2, 40, 45, 22]. In particular, the Direct Hashing and Pruning algorithm [40] partitions itemsets into buckets and prunes buckets that violate the minimum support constraint. The Partition Algorithm [45] partitions the transactions database and processes one partition at a time. These partitioning approaches depend on the assumption that transactions independently contribute to the itemset counts in a transactions database. In a first-order knowledge base, the facts are interconnected by the rules and arguments. This dependency will be lost if we directly partition the knowledge base in the state-of-the-art ways. In our approach, we preserve data dependency by relaxing the non-overlapping requirement and designing a new algorithm that partitions the knowledge base into independent but possibly overlapping partitions.

Mining frequent subgraphs. The rule mining problem is closely related to mining frequent subgraphs [17, 56, 29, 28], where the knowledge base is represented as a single directed graph. The major difference is the notion of a subgraph: In the first-order rule mining problem, a subgraph is a graph pattern where the edges are labeled but vertices are variables. This leads to an important consequence we address in this paper: A more sophisticated counting algorithm is needed to count the frequencies of *grounded* subgraphs [43], namely, subgraphs with variables substituted by constants. The grounding problem of parameterized subgraphs is not addressed in the frequent subgraph mining literature.

3. FIRST-ORDER MINING PROBLEM

We study the problem of mining first-order inference rules from web-scale knowledge bases: Given a knowledge base of (subject, predicate, object) (or (s, p, o)) triples, we mine first-order Horn clauses of form

$$(\mathbf{w},\ \mathbf{B} \rightarrow H(x, y)), \tag{1}$$

where the body $\mathbf{B} = \bigwedge_i B_i(\cdot, \cdot)$ is a conjunction of predicates, H is the head predicate, and $\mathbf{w}$ is a scoring metric reflecting the likelihood of the rule being true.

As in AMIE [20] and other ILP systems [36, 49], we use a language bias and assume the Horn clauses to be connected and closed. Two atoms are connected if they share a variable. A rule is *connected* if every atom is connected transitively to every other atom in the rule. A rule is *closed* if every variable appears at least twice in different predicates. This assumption ensures that the rules do not contain unrelated atoms or variables.

[1] http://dsr.cise.ufl.edu/projects/probkb-web-scale-probabilistic-knowledge-base.

Our primary focus of this paper is *scalable mining*. Previous approaches [20, 47] scale to knowledge bases of 1 million facts, but no existing work mines inference rules from the 112 million entities and 388 million facts of Freebase. We investigate a series of parallelization and optimization techniques to achieve this scale, and we study how to use state-of-the-art data processing systems, e.g., Spark, to efficiently implement the parallel algorithms.

The first-order mining problem has similarities with association rule mining in transactions databases [20], but they are essentially different. In first-order rules, the atoms are *parameterized* predicates. Each parameterized predicate can be *grounded* to a set of ground atoms. Depending on the size of the knowledge base, each rule can have a large number of possible ground instances. This makes mining first-order knowledge more challenging than mining traditional association rules in transactions databases.

3.1 Scoring Metrics

We review the support and confidence metrics for first-order Horn clauses. They have counterparts in association rule mining in transactions databases, but are different from them as discussed above.

Support. The *support* of a rule is defined to be the number of distinct pairs of subject and object in the head of all instantiations that appear in the knowledge base:

$$\text{supp}(\mathbf{B} \rightarrow H(x,y)) \coloneqq \#(x,y) : \exists z_1, \ldots, z_m : \mathbf{B} \wedge H(x,y). \quad (2)$$

In Equation (2), $\mathbf{B}$ and H denote the body and head, respectively. $z_1, \ldots, z_m$ are the variables of the rule in addition to x and y.

Confidence. The *confidence* of a rule is defined to be the ratio of its predictions that are in the knowledge base:

$$\text{conf}(\mathbf{B} \rightarrow H(x,y)) \coloneqq \frac{\text{supp}(\mathbf{B} \rightarrow H(x,y))}{\#(x,y) : \exists z_1, \ldots, z_m : \mathbf{B}}. \quad (3)$$

For each rule, we compute its support and confidence, and set $\mathbf{w} = (\text{supp}, \text{conf})$ in (1). The support and confidence metrics together determine the quality of a rule. [20, 47] introduces other scoring functions, e.g., head coverage, PCA confidence, statistical relevance. These metrics can be implemented using the same principle as we propose in this paper.

4. PRELIMINARIES

Before describing the OP algorithm in detail, we introduce the notion of knowledge bases, their relational model, and Spark.

4.1 Knowledge Bases

In this paper, we model a *knowledge base* as a collection of (subject, predicate, object) facts $\Gamma = \{(s,p,o)\}$ with the following semantics:

- Each subject or object refers to a real-world *entity*. For example, Barack Obama, USA are well-known entities.
- Each predicate specifies relationships among its subjects and objects. We call each relationship a *fact*. For example, the fact (Barack Obama, wasBornIn, USA) specifies that Barack Obama was born in the USA.
- Each entity belongs to one or more types. Correspondingly, each predicate specifies a binary relation between one or more pairs of types. For example, the wasBornIn predicate specifies a binary relation between Person and Place. We call these types *domain* and *range* of the predicate, respectively.
- We call the predicates, along with their domains and ranges, the *schema* of the knowledge base.

Example 1. As a concrete example, in Freebase, the predicates are called *properties*. Each property belongs to a specific type, written as domain/type/property, where a domain denotes a manual division of Freebase consisting of related types[2]. Thus, a predicate "film/film/country(x,y)" means film x is produced in country y. The inference rules are interpreted accordingly, illustrated below:

$$\text{film/film/sequel}(x,z),\ \text{film/film/country}(z,y) \rightarrow \text{film/film/country}(x,y).$$

The rule states that if a film x has a sequel z, and the sequel z is produced in country y, then film x is produced in country y. □

4.2 Relational Knowledge Base Model

[12] introduces a relational model for probabilistic knowledge bases. In this section, we review the model for inference rules. This relational representation allows us to compactly store inference rules and apply them in batches using join queries. The main challenge with inference rules is that they have flexible structures, conflicting with the relational tables with fixed columns. To adapt for their structures, we use the concept of structural equivalence so that each equivalent class has a fixed table format.

Definition 1. Two first-order clauses are defined to be *structurally equivalent* if they differ only in entities, types, and predicates.

It is verifiable that the structural equivalence relation in Definition 1 is an equivalence relation. Thus, first-order clauses can be divided into equivalent classes accordingly. Since the first-order clauses in each equivalent class differ only in entities, types, and predicates, it suffices to use these elements to identify inference rules in an equivalence class. In other words, each equivalent class can be stored in a fixed-column table, with the columns being entities, types, and predicates.

Example 2. Consider the following inference rules:

1. $\forall x \in$ Person, $\forall y \in$ Person:
   isMarriedTo$(x, y) \rightarrow$ isMarriedTo(y, x);
2. $\forall x \in$ Book, $\forall y \in$ Person:
   influences$(x, y) \rightarrow$ isInterestedIn(y, x);
3. $\forall x \in$ Person, $\forall y \in$ Actor, $\forall z \in$ Movie:
   directed(x, z), actedIn$(y, z) \rightarrow$ influences(x, y);
4. $\forall x \in$ Person, $\forall y \in$ Actor, $\forall z \in$ Org:
   worksAt(x, z), worksAt$(y, z) \rightarrow$ influences(x, y).

Rules 1 and 2 are structurally equivalent since their only differences are the predicates (isMarriedTo, influences, isInterestedIn) and types (Person, Book). Similarly, Rules 3 and 4 are structurally equivalent. Therefore, we store Rules 1 and 2 in one table with the columns specifying the predicates of the head and body, as shown in Table 1 (left). We store Rules 3 and 4 in Table 1 (right), its columns storing the head and first, second predicates of the rule body. We have omitted the types since they are not involved in the join queries to evaluate the rules, as we discuss in Section 5.2.

Head	Body
isMarriedTo	isMarriedTo
isInterestedIn	influences

Head	Body1	Body2
influences	directed	actedIn
influences	worksAt	worksAt

Table 1: (Left) Relational table for rules 1 and 2. (Right) Relational table for rules 3 and 4.

The rules in one table are interpreted in the same way. For example, both rules in Table 1 (left) mean "if Body(x,y) holds, then Head(x,y) holds". By classifying the rules into structurally equivalent classes, we are able to store them into fixed-column tables and perform relational queries to evaluate them in batches. □

[2]In Freebase, domains are used to conceptually organize the types. We do not use this terminology elsewhere in the paper.

R1: $\text{Person}_1 \xrightarrow{\text{isMarriedTo}} \text{Person}_2 \xrightarrow{\text{livesIn}} \text{City} \xleftarrow{\text{livesIn}} \text{Person}_1$	$\text{isMarriedTo}(P_1, P_2), \text{livesIn}(P_2, C) \rightarrow \text{livesIn}(P_1, C)$
R2: $\text{Person} \xrightarrow{\text{directed}} \text{Movie} \xleftarrow{\text{actedIn}} \text{Actor} \xleftarrow{\text{influences}} \text{Person}$	$\text{directed}(P, M), \text{actedIn}(A, M) \rightarrow \text{influences}(P, A)$
R3: $\text{Person} \xrightarrow{\text{worksAt}} \text{Organization} \xleftarrow{\text{worksAt}} \text{Actor} \xleftarrow{\text{influences}} \text{Person}$	$\text{worksAt}(P, O), \text{worksAt}(A, O) \rightarrow \text{influences}(P, A)$

Figure 1: (Left) Cycles from Example 3. The first and last nodes in R1-R3 denote the same start and end node in the cycle. (Right) Corresponding rules.

4.3 Spark Basics

Spark is a cluster computing framework. Based on its core idea of resilient distributed datasets (RDDs)–read-only collections of objects partitioned across a set of machines–it defines a set of parallel operations. Using these operations, Spark allows users to express a rich set of computation tasks. The operations we use in this paper are listed below:

- `map/flatMap` Transforms an RDD to a new RDD by applying a function to each element in the input RDD.
- `groupByKey` Transforms an RDD to a new RDD by grouping by a user-specified key. In the result RDD, each key is mapped to a list of values of the key.
- `reduceByKey` Transforms an RDD to a new RDD by grouping by a user-specified key and performs a reduce function to the values of each key. In the result RDD, each key is mapped to the result value of the reduce function.

In our parallel rule mining algorithm, we represent the set of facts $\{(s, p, o)\}$ and the set of rules $\{(h, b_1, b_2)\}$ as two RDDs and express the algorithm using the above parallel operations.

5. ONTOLOGICAL PATHFINDING

The Ontological Pathfinding (OP) algorithm is similar to relational pathfinding [42], but instead of finding paths over an entity-relationship graph, the OP algorithm searches the domain-range schema graph. The schema graph constrains the search space to syntactically correct rules according to the arguments of each predicate. After constructing candidate rules, OP partitions them into smaller independent parts. It accepts a maximum size argument s, a maximum rules size m, and partitions the candidate rules, making a best effort to satisfy the size requirements. Finally, OP prunes candidate rules with respect to the functional constraint t and evaluates the rule batches using the parallel mining algorithm.

Algorithm 1 summarizes the major steps of Ontological Pathfinding to mine inference rules from a knowledge base Γ. We explain each step in detail in the following sections.

Algorithm 1 Ontological Pathfinding(Γ, t, s, m)

```
M ← construct-rules(Γ.schema)
{(Γ_i, M_i)} ← partition(Γ, ∅, M, s, m)
rules ← ∅
for all (Γ_i, M_i) do
    M_i ← prune(Γ_i, M_i, t)
    rules ← rules ∪ parallel-rule-mining(Γ_i, M_i)
return rules
```

5.1 Rule Construction

As discussed in Section 4.1, all entities and predicates are *typed* in a knowledge base, i.e., each entity belongs to one or more types, and each predicate has a domain and a range. Typing restricts the predicates that can be combined to form candidate rules: they must have corresponding types for the joining variables. In this section, we utilize the schema to construct candidate rules.

Definition 2. The *schema graph* of a knowledge base is defined to be a graph $G_\Gamma = (V, E)$, where $V = \{v_1, \ldots, v_{|V|}\}$ is the set of nodes representing types and $E = \{e_p(v_i, v_j)\}$ is the set of labeled directed edges representing typed predicates p with v_i as the domain and v_j as the range.

In a web-scale knowledge base like YAGO and DBPedia, we often have a *type hierarchy* to divide types into subtypes for finer classification. For instance, the "Person" type has subtypes including "Actor", "Professor". These types are related by the "subClassOf" edges. In Figure 2, we have a "subClassOf" edge from "Actor" to "Person", indicating that "Actor" is a subtype of "Person". To construct candidate rules, the subclasses need to inherit all the edges from their ancestors. The inherited edges are defined in Definition 3 using the following notations: 1. $A(v)$ denotes the ancestor nodes u of v such that there is a path from v to u with all edges labeled as "subClassOf"; and 2. $E(u/v)$ denotes the neighboring non-subClassOf edges of u with at least one u substituted by v.

Definition 3. We define the *schema closure graph* of a schema graph $G = (V, E)$ to be $G' = (V, E \cup E')$ where

$$E' = \bigcup_{v \in V, u \in A(v)} E(u/v).$$

Example 3. In Table 2, we provide an example schema from the YAGO knowledge base. Its schema graph is shown in Figure 2. In this graph, each type in the domain and range columns is represented as a node, and each row in Table 2 is represented as an edge from domain to range. Since Actor is a subclass of Person, we infer additional edges by having Actor inherit Person's non-subClassOf edges, as shown by dashed arrows in Figure 2. □

Predicate	Domain	Range
livesIn	Person	City
isMarriedTo	Person	Person
worksAt	Person	Organization
directed	Person	Movie
influences	Person	Actor
actedIn	Actor	Movie
subClassOf	Actor	Person

Table 2: Example KB schema from YAGO knowledge base.

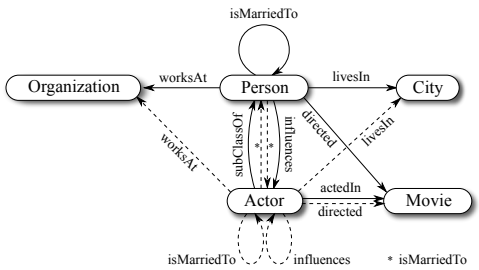

Figure 2: Example schema closure graph. Dashed arrows indicate inherited edges.

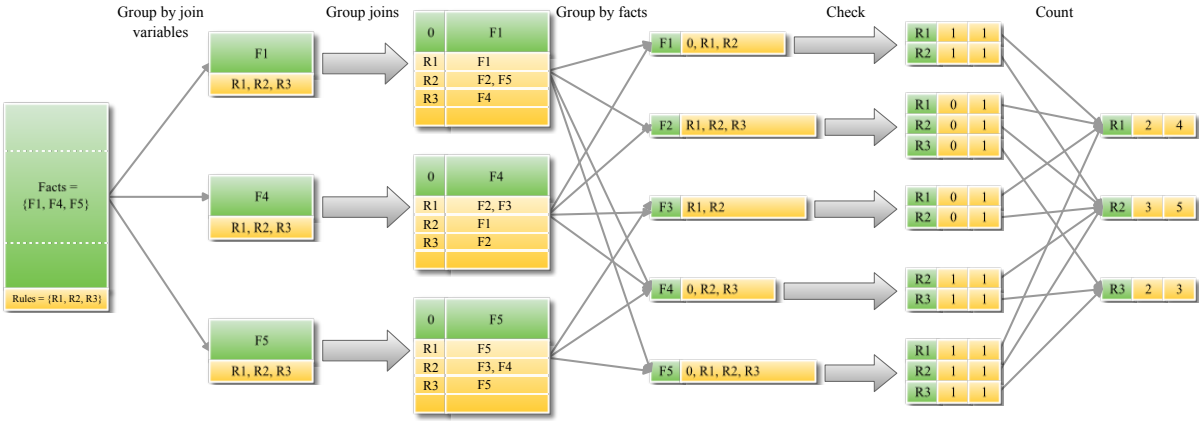

Figure 3: Parallel rule mining: KB divided into groups by join variables, each group running GROUP-JOIN to apply inference rules.

Following Definition 3, Algorithm 2 shows how to compute the closure of a specific node v in a schema graph G by DFS. When visiting a node v, we recursively visit its ancestors (Line 3) and add their neighboring edges to v (Line 4). Visiting each unvisited node in G yields its schema closure graph.

Algorithm 2 Closure$(G = (V, E), v)$

1: **if** !v.visited **then**
2: **for all** $e_{\text{subClassOf}}(v, u) \in E$ **do**
3: Closure(G, u)
4: $E \leftarrow E \cup E(u/v)$
5: v.visited $\leftarrow$ True

By traversing the closure graph, we identify a syntactically correct rule when we detect a cycle. For instance, Figure 1 (Left) shows three example cycles from Example 3, and Figure 1 (Right) shows the corresponding rules constructed from the cycles. Although we have directed edges in the schema graph, we traverse it in an undirected manner: from any vertex v, we visit its neighbors from both incoming and outgoing edges. R3 shows an example path containing an inherited edge, Actor $\xrightarrow{\text{worksAt}}$ Organization, inherited from Actor's ancestor, Person.

5.2 Parallel Rule Mining

Given the relational knowledge base model in Section 4.2, we store candidate inference rules in tables, allowing us to apply them in batches by joining the rules and facts tables. Furthermore, we design a parallel rule mining algorithm that divides the knowledge base into groups running parallel in-memory group joins. We introduce the algorithm with the following equivalent class of rules:

$$p(x, y) \leftarrow q(x, z), r(y, z).$$

In Section 5.2.1, we generalize it to other rule classes. We present Algorithm 3 using Spark primitives, but we note that it is a general parallel algorithm consisting of basic parallel operations, conforming to the MapReduce paradigm. Figure 3 illustrates how Algorithm 3 transforms datasets using parallel operations.

In Step 1, we map each (s, p, o), i.e., (subject, predicate, object), triple to a pair $(o, (p, s))$, o serving as a key for the GroupByKey operation that follows. In Step 2, the GroupByKey operation groups all $(o, (p, s))$ tuples with the same object o into one group. This ensures the tuples with the same joining variable (o) are in the same group. In addition, we broadcast the rule table to each group to ensure all relevant data are collocated for the joins. Consequently, the groups run disjoint in-memory group-joins, Algorithm 4, and are executed in parallel.

Algorithm 3 Parallel Rule Mining

Require: `facts` = {(`pred`, `sub`, `obj`)}
Require: `rules` = {(`ID`, `head`, `body1`, `body2`)}
1: **`Map`** each fact (`pred`, `sub`, `obj`) $\in$ `facts` to (`obj`, (`pred`, `sub`)).
2: **`GroupByKey`** `obj`, yielding a list of {(`pred`, `sub`)} pairs for each `obj`.
3: **`FlatMap`** the (`obj`, {(`pred`, `sub`)}) pairs to GROUP-JOIN(`obj`, {(`pred`, `sub`)}, `rules`), using Algorithm 4, yielding a list of {(`pred`, `sub`, `obj`), `rule.ID`)} pairs.
4: **`ReduceByKey`** (`pred`, `sub`, `obj`), deduplicating the `rule.ID` s for each (`pred`, `sub`, `obj`) triple.
5: **`FlatMap`** the ((`pred`, `sub`, `obj`), {`rule.ID`}) tuples to CHECK({`rule.ID`}), using Algorithm 5, yielding a list of {`rID`, (`correct`, `1`)}pairs.
6: **`ReduceByKey`** `rule.ID`, summing the `correct` and `1` values.
7: **`Map`** each (`rule.ID`, (`sum`, `count`)) to (`rule.ID`, `sum/count`) pairs.

The GROUP-JOIN algorithm is an in-memory hash-join. Since the join variable o is matched by the previous GroupByKey, GROUP-JOIN matches predicates to the rule body. It starts by hashing the input list of (p, s) pairs by predicate p (Line 3), creating a list of subjects for each p. This hash table is used to retrieve joined entities to be output as the subject and object as inferred by the rule. For each rule, its body predicates match two lists of subjects from the hash table. The algorithm infers the rule head with combinations of subjects from the two matched lists (Lines 4-7).

In the GROUP-JOIN algorithm, we output the input facts (Line 2) and the inferred facts (Line 7). Each triple is output as a key, with the value being the positive ID of the rule if it is inferred by that rule, or 0 if it is from the input knowledge base. These IDs are used to verify the inference results: if a fact is associated with an ID of 0, it exists in the input knowledge base; otherwise, it is inferred by inference rules specified by the ID list.

Algorithm 4 GROUP-JOIN(`obj`, `ps` = {`pred`, `sub`}, `rules`)

1: **for all** (`pred`, `sub`) $\in$ `ps` **do**
2: `emit` ((`pred`, `sub`, `obj`), `0`)
3: `preds` $\leftarrow$ `ps.groupBy` (`pred`)
4: **for all** $r \in$ `rules` **do**
5: **for all** `sub1` $\in$ `preds.get` (r.`body1`) **do**
6: **for all** `sub2` $\in$ `preds.get` (r.`body2`) **do**
7: `emit` ((r.`head`, `sub1`, `sub2`), r.`ID`)

In Step 4, we group by the output facts, each group computing a list of rule IDs that generate the fact. Algorithm 5 processes this list and determines if each rule is generating a correct fact by searching

for 0 in the list. It outputs a (ID, (c, 1)) tuple for each rule, where c indicates the correctness of the inferred fact. Thus, the component-wise sum of the (c, 1) tuples for each rule is the number of correct and total facts, respectively, inferred by the rule. Steps 6 and 7 group by the rules, sum the correct and total counts, and compute the confidence of each rule.

Algorithm 5 CHECK(`rs = {rule.ID}`)

```
1: correct ← rs.contains(0)
2: for all rule.ID ∈ rs do
3:    emit(rule.ID, (correct, 1))
```

The correctness of Algorithm 3 follows from the fact that the entire set of rules is broadcast to each group and that the groups are disjoint from each other (recall that they are grouped by key o). Therefore, Step 3 properly applies rules to the facts. In Step 5, Algorithm 5 generates individual correct (0 or 1) and total (1) counts for rules inferring each fact. Since each fact contains the "0" flag to determine its correctness, aggregating the results from all facts generates final correct and total counts for each rule.

5.2.1 General Rule Mining

To generalize Algorithm 3 to other rule classes, we recall from Section 3 that the Horn clauses are assumed to be *connected* and *closed*. Thus, for a general rule with rule ID $\dot{r}$:

$$h(x, y) \leftarrow b_1, b_2, \ldots, b_k,$$

we can arrange the body atoms so that each b_i is connected to b_{i+1}, $i = 1, \ldots, k-1$, and b_k is connected to $h(x, y)$, by a shared variable $\mathbf{z_i}$. The general rule mining algorithm, Algorithm 6, allows $\mathbf{z_i}$ to be vectors and to contain repeated variables.

Algorithm 6 General Rule Mining

```
Require: facts = {(p, x, y)}
Require: rules = {(ṙ, h, b_1, ..., b_k)}
1: j_1 ← facts
2: for all pairs (b_i, b_{i+1}) with shared variable z_i do
3:    j_i ← j_i.GroupByKey(z_i)
4:    f_{i+1} ← facts.GroupByKey(z_i)
5:    if i + 1 < k then
6:       j_{i+1} ← {(z_{i+1}, (ṙ, x_{i+1}))} =
            GROUP-JOIN(j_i, f_{i+1}, rules)
7:    else
8:       j_k ← {((h, x, y), ṙ)} = GROUP-JOIN-LAST(j_i, f_k, rules)
9: Process join result j_k, as in Steps 4-7 of Algorithm 3.
```

Algorithm 6 joins two body atoms at a time. j_i denotes the result of joining $b_1, \ldots, b_i$, and is used as the operand for joining the next rule body, f_{i+1}, in each iteration. The GROUP-JOIN and GROUP-JOIN-LAST methods in Lines 6 and 8 implement the rule semantics. GROUP-JOIN performs the join and outputs the tuples with the next shared variable $\mathbf{z_{i+1}}$ as the key, along with the rule ID, $\dot{r}$, and any variables referred to by the head or subsequent body atoms. GROUP-JOIN-LAST completes the join and infers facts from the rules, generating a list of (fact, rule ID) pairs as in Algorithm 4 to be further processed to evaluate the confidence of each rule, as in Steps 4-7 of Algorithm 3.

As a remark, we note that Algorithm 6 requires the input rules be of the same form. Thus, if there are N equivalent classes (defined in Section 4.2) of rules, we need N rule tables and N calls of Algorithm 6 to complete the mining task. On the other hand, each run of Algorithm 6 is highly efficient and optimized as we apply the rules in batches and in parallel. Thus, our approach trades off between generality and efficiency.

5.3 Rule Pruning

One performance barrier we observe in the candidate rules is that some of them generate prohibitively large intermediate results due to large-degree variables in the joining predicates, as demonstrated by the following example:

$$\text{diedIn}(x, z), \text{wasBornIn}(y, z) \rightarrow \text{hasAcademicAdvisor}(x, y) \tag{4}$$

In the above rule, we have variable z as the joining variable. Meanwhile, in places with large populations, e.g., New York City, the California state, etc, there may be hundreds of thousands of people who were born or dead. Rules like this are untrustworthy, as most of the intermediate results are irrelevant enumerations of the involving arguments (x, y in Rule (4)).

This large join problem is not uncommon among candidate rules as they are constructed from the KB schema with no validation using the facts. Thus, rules can accidentally have irrelevant predicates that coincide with the join variable with a large degree. To detect these rules, we use histograms to determine the functional property of inference rules:

- Predicate Histogram $H^0 = \{(p, |\{p(\cdot, \cdot)\}|)\}$;
- Predicate-Subject Histogram $H^1 = \{(p, x, |\{p(x, \cdot)\}|)\}$;
- Predicate-Object Histogram $H^2 = \{(p, y, |\{p(\cdot, y)\}|)\}$.

In a functional notation, we write $H^0(p) = |\{p(\cdot, \cdot)\}|$, $H^1(p, x) = |\{p(x, \cdot)\}|$, and $H^2(p, y) = |\{p(\cdot, y)\}|$. H^0 is used to compute the size of a KB partition, as explained in Section 6. H^1 and H^2 determine the size of intermediate results. For instance, the size of the join for Rule (4) can be computed by

$$\sum_z H^2(\text{diedIn}, z) \cdot H^2(\text{wasBornIn}, z).$$

In the definition below, we omit the position descriptors (H^1, H^2) and use $H(p, z)$ to denote the histogram entry for predicate p and join variable z in a general rule, the position of z determined by the join under consideration.

Definition 4. For a connected, closed rule r:

$$h \leftarrow b_1, \ldots, b_l,$$

we define the *non-functionality* of r as

$$\text{NF}(r) = \max_{b_i, b_j \text{ connected by } z} \{\min(H(b_i, z), H(b_j, z))\}. \tag{5}$$

A *functional constraint* t accepts rules r with $\text{NF}(r) \leq t$.

According to Definition 4, a functional constraint requires each join, represented by a pair of connected atoms of the rule, have a join variable z with no more than t joined instances as determined by $\min(H(b_i, z), H(b_j, z))$, z ranging over all joined values. We use non-functionality as an empirical indication of rule correctness. Viewing a knowledge graph as a sparse natural graph–e.g., Freebase and YAGO2s has a sparsity of 3.11×10^{-8} and 9.82×10^{-7}, respectively–we justify our approach by the empirical power-law degree distribution of natural graphs that only a few entities in a knowledge graph have a large degree [21], implying that neighbors of large degree entities are unlikely inter-connected with one another as suggested by the non-functional joins.

In our experiment, we observe that violations of functional constraints are strong indications of incorrectness of the rules, and more than 99% of the non-functional rules are wrong. Removing those erroneous rules improves performance and rule quality. By varying the constraint t, we show that a reasonable choice lies between 50 and 250 and set $t = 100$ in the default configuration. We experimentally justify our choice in Section 7.4.

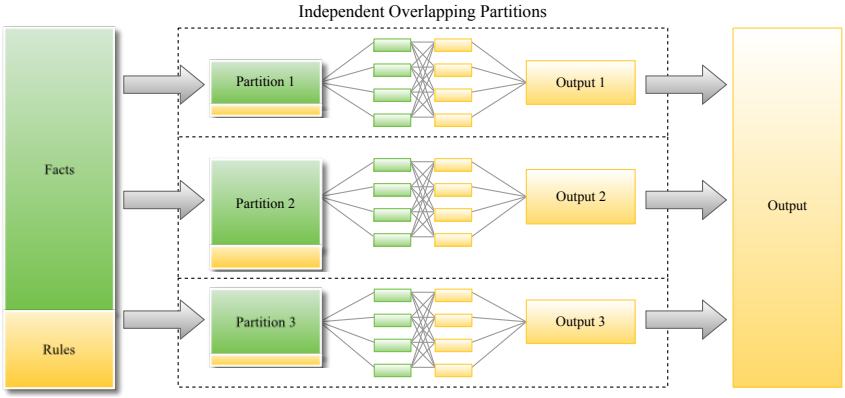

Figure 4: Partitioning algorithm: KB partitioned into smaller overlapping parts running independent mining algorithm instances.

Example 4. As an example, Table 3 shows a histogram from the YAGO2 knowledge base containing the number of people born and died in New York City, London, and Montreal.

wasBornIn	NYC	1287
wasBornIn	London	1584
wasBornIn	Montreal	618
diedIn	NYC	737
diedIn	London	951

Table 3: Histogram for "wasBornIn" and "diedIn".

Using this histogram, we determine the non-functionality of Rule (4) to be 951, the joining variable z being NYC and London. The total number of facts inferred by Rule (4) is $1287 \times 737 + 1584 \times 951 = 2\,454\,903$, 734 times larger than the head predicate hasAcademicAdvisor ($H^0(\text{HasAcademicAdvisor}) = 3340$). □

Analysis. To analyze Algorithm 3, we study the size of its intermediate result, which is dominated by joining the rule body. Suppose we have facts tables S, T, rules table M, and a functional constraint t. We denote the histograms by H_S and H_T, the position of the join variable implied by context of the rule. We use $M[\cdot], \hat{M}[\cdot]$ to denote the projection and distinct projection of table M to the specified columns, respectively. The join size is then given by:

$$
\begin{aligned}
&\sum_z \sum_{(p,q)\in M[b_1,b_2]} H_S(p,z)\cdot H_T(q,z)\\
\le& t\sum_z \sum_{(p,q)\in M[b_1,b_2]} \max\{H_S(p,z), H_T(q,z)\}\\
\le& t\sum_z \sum_{(p,q)\in M[b_1,b_2]} (H_S(p,z)+H_T(q,z))\\
=& t\left(\sum_z \sum_{(p,q)\in M[b_1,b_2]} H_S(p,z) + \sum_z \sum_{(p,q)\in M[b_1,b_2]} H_T(q,z)\right)\\
\le& t\left(|M|\sum_z \sum_{p\in\hat{M}[b_1]} H_S(p,z) + |M|\sum_z \sum_{q\in\hat{M}[b_2]} H_T(q,z)\right)\\
\le& t(|M||S| + |M||T|)\\
=& t|M|(|S|+|T|),
\end{aligned}
\tag{6}
$$

where the first inequality follows from Definition 4. Hence, The time complexity of Algorithm 3 is dominated by $O(t|M|(|S| + |T|))$. In case of a self-join, we have $S = T$, and the complexity reduces to $O(t|M||S|)$. More generally, for longer rules, the complexity is $O(t^{l-1}|M||S|)$, where l is the body length of the rule. In practice, as we show in Section 7.4, a reasonable t lies between 50-250. Comparing with a direct join of facts and rules, $O(|M||S|^l)$, we achieve a notable improvement with pruning.

6. PARTITIONING

Even if we parallelize the rule mining algorithm, it still suffers from data shuffling between stages when the knowledge base expands in scale. In this section, we introduce another level of partitioning that divides the KB into a set of independent but possibly overlapping partitions running smaller instances of the rule mining algorithm, as shown in Figure 4.

The partitioning algorithm improves performance by accepting a size constraint and returns partitions that satisfy the constraint. The smaller partitions are more efficiently processed than the entire KB. We begin by introducing the notions of Γ-partition and Γ-size. They allow us to determine the size of a partition. We utilize this information to assign each rule to an appropriate partition.

Definition 5. Denote the predicates in rule $r = p_0 \leftarrow p_1, \ldots, p_l$ by $\Gamma(r) = \{p_0, p_1, \ldots, p_l\}$. We define the Γ-*partition*, with respect to rules $M_i = \{r_1, \ldots, r_m\}$, to be the set of predicates

$$\Gamma(M_i) = \bigcup_{i=1}^{m} \Gamma(r_i).$$

Given an input KB Γ and partitioned rules $M = \{M_1, \ldots, M_k\}$. Let $\Gamma_i = \{p(x,y) \in \Gamma | p \in \Gamma(M_i)\}$ be the partition of facts induced by M_i. Then Γ_i contains all facts we need to evaluate rules $r \in M_i$. Thus, (Γ_i, M_i) defines an *independent* partition of the input KB. Running Algorithm 6 on (Γ_i, M_i) returns the mining results for rules M_i. The size of the partition indicates how long the algorithm runs and can be determined using the predicate histogram H^0 from Section 5.3.

Definition 6. We define the Γ-*size* of a rule set M_i with respect to KB Γ to be

$$\sigma(\Gamma, M_i) = |\Gamma_i| = \sum_{p\in\Gamma(M_i)} H^0(p), \tag{7}$$

where $H^0 = \{(p, |\{p(\cdot,\cdot)\}|)\}$ is the predicate histogram.

Example 5. Consider the knowledge base Γ and rule table M in Figure 5. M is partitioned into two parts, M_1 and M_2 (initially without the last row, r). According to Definition 5, we have:

$$\Gamma(M_1) = \{P_1, Q_1, R_1, P_2, Q_2, R_2\},$$
$$\Gamma(M_2) = \{P_3, Q_3, R_3, P_4, Q_4, R_4\}.$$

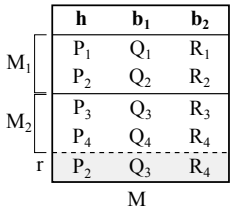

Figure 5: Rule table M, initial partitions M_1, M_2, and unpartitioned rule r.

Assuming each predicate has 1 fact, namely, $H^0(P_i) = H^0(Q_i) = H^0(R_i) = 1$, we have $\sigma(\Gamma, M_1) = \sigma(\Gamma, M_2) = 6$.
Now consider the addition of rule $r = (P_2, Q_3, R_4)$. We have:

$$\Gamma(M_1 \cup \{r\}) = \{P_1, Q_1, R_1, P_2, Q_2, R_2, Q_3, R_4\},$$
$$\Gamma(M_2 \cup \{r\}) = \{P_3, Q_3, R_3, P_4, Q_4, R_4, P_2\}.$$

Consequently, $\sigma(\Gamma, M_1 \cup \{r\}) = 8, \sigma(\Gamma, M_2 \cup \{r\}) = 7$. Adding r to M_2 incurs a smaller increase of size than adding it to M_1. □

Given an upper bound of the Γ-size s and the number of rules m for each partition, our goal is to find a partition $\{M_1, \ldots, M_k\}$ of M that satisfies the following constraints:

$$\text{(C1)} \quad \sigma(\Gamma, M_i) \le s, \qquad 1 \le i \le k$$
$$\text{(C2)} \quad |M_i| \le m, \qquad 1 \le i \le k$$
$$\text{(C3)} \quad \bigcup_{i=1}^{k} M_i = M,$$
$$\text{(C4)} \quad M_i \cap M_j = \varnothing, \qquad 1 \le i < j \le k$$

We seek to find a partition $\{M_1, \ldots, M_k\}$ with as a small k as possible. Without a priori knowledge of the optimal k, we use a recursive binary partitioning scheme in Algorithm 7. In each recursive step, the input rule set M is partitioned into two smaller parts, M_1 and M_2. The algorithm terminates when all partitions satisfy the size constraints (C1) and (C2). The partitions satisfy the completeness and disjointness constraints (C3) and (C4) at each recursive step as M is partitioned into M_1 and M_2 by Algorithm 8 such that $M_1 \cup M_2 = M$ and $M_1 \cap M_2 = \varnothing$.

Algorithm 7 RecPartition(Γ, Π, M, s, m)

```
1: if σ(Γ, M) ≤ s and |M| ≤ m then
2:     Π ← Π ∪ {M}
3: else
4:     (M1, M2) ← 2-Partition(Γ, M)
5:     if M1 = ∅ then
6:         Π ← Π ∪ {M2}
7:     else if M2 = ∅ then
8:         Π ← Π ∪ {M1}
9:     else
10:        RecPartition(Γ, Π, M1, s, m)
11:        RecPartition(Γ, Π, M2, s, m)
```

Specifically, we first determine whether to partition the input M by checking if it already satisfies the size constraints (C1) and (C2) (Line 1). If it does, we add it to the final set of partitions and return (Line 2); otherwise, we use a binary partitioning algorithm to partition the rules (Line 4) and recursively partition the sub-parts (Lines 10-11). Lines 5-8 handle special cases where the size constraint (C1) cannot be satisfied. This happens when $s < H^0(p)$ for some predicate p. The binary partitioning algorithm is described in Algorithm 8, using a greedy assignment strategy: each rule is assigned to the partition with a smaller increase in size (Lines 4-9). Note that in Lines 4-5, we add a penalty term $p(\Gamma, M)$ to penalize the larger partition, preventing it from absorbing all subsequent

Algorithm 8 2-Partition(Γ, M)

```
1: M1 ← ∅
2: M2 ← ∅
3: for all r ∈ M do
4:     Δ1 ← σ(Γ, M1 ∪ {r}) − σ(Γ, M1) + p(Γ, M1)
5:     Δ2 ← σ(Γ, M2 ∪ {r}) − σ(Γ, M2) + p(Γ, M2)
6:     if Δ1 < Δ2 then
7:         M1 ← M1 ∪ {r}
8:     else
9:         M2 ← M2 ∪ {r}
10: return (M1, M2)
```

rules, as duplicate predicates do not increase the partition size. In our experiments, we set $p(\Gamma, M) = \frac{1}{50}\sigma(\Gamma, M)$.

Analasis. Algorithm 7 makes a best effort to satisfy the size requirement s. If $H^0(p) > s$ for some predicate p, Algorithm 7 returns in Lines 5-8 without partitioning the predicate. Assuming Algorithm 7 results in N partitions $\{M_1, \ldots, M_N\}$, the size of the largest partition being s_m, then the overall time complexity to evaluate the partitioned facts and rules tables, based on (6), is:

$$\sum_{i=1}^{N} O(t^{l-1} s_m |M_i|) = O\left(t^{l-1} s_m \sum_{i=1}^{N} |M_i|\right) = O(t^{l-1} s_m |M|), \tag{8}$$

where t is the functional constraint and l is the length of the rule body as in Section 5.3. The time complexity is bounded with respect to the largest partition instead of the input knowledge base. Thus, partitioning reduces time complexity from $O(t^{l-1}|\Gamma||M|)$ to $O(t^{l-1} s_m |M|)$, allowing us to control the complexity by tuning the size constraints for very large knowledge bases.

Since Γ-partitions are induced from partitions of inference rules (Definition 5) and rules from different partitions may contain duplicate head or body predicates, the Γ-partitions may overlap, as illustrated by the addition of rule r in Example 5. To measure this overlap, we introduce the notion of *degree of overlap*.

Definition 7. The *degree of overlap* (DOV) of a set of rule parts $M = \{M_1, M_2, \ldots, M_k\}$ is defined to be

$$\mathrm{DOV}(M) = \frac{\sum_i \sigma(\Gamma, M_i)}{\sigma\left(\Gamma, \bigcup_i \Gamma(M_i)\right)}. \tag{9}$$

In Equation (9), the numerator $\sum_i \sigma(\Gamma, M_i)$ is the total size of the partitions we make to evaluate the inference rules. The denominator $\sigma\left(\Gamma, \bigcup_i \Gamma(M_i)\right)$ is the size of the KB induced by rules M. A DOV greater than 1 indicates overlapping partitions.

Example 6. Consider the partitioning scheme from Example 5. While the initial partitions M_1 and M_2 are disjoint, adding r to M_1 or M_2 results in overlapping partitions:

$$\mathrm{DOV}(\{M_1, M_2\}) = \frac{6+6}{12} = 1;$$
$$\mathrm{DOV}(\{M_1, M_2 \cup \{r\}\}) = \frac{6+7}{12} = 1.0833;$$
$$\mathrm{DOV}(\{M_1 \cup \{r\}, M_2\}) = \frac{8+6}{12} = 1.1667. \qquad \square$$

Partition Independence. We conclude this section by remarking the difference between partitioning in Algorithm 7 and parallelization in Algorithm 3. Algorithm 7 breaks the input knowledge base into smaller independent partitions so that each partition runs its own instance of Algorithm 3. Algorithm 3 divides the input knowledge base into correlated groups running sub-procedures of a sin-

gle mining task. Algorithm 3 is used by Algorithm 7 as the parallel mining algorithm in each partition. The two-level partitioning-parallelization scheme is elucidated in Figure 4. These techniques combined scale the rule mining algorithm to Freebase.

7. EXPERIMENTS

We validate our approaches by mining inference rules from YAGO and Freebase. Our work contributes the first rule set for Freebase– 36,625 first-order inference rules. In this section, we present our results, compare with the state-of-the-art KB rule mining algorithm, AMIE [21], and analyze individual techniques from Sections 5 and 6. We begin by describing the datasets and experiment setup.

YAGO. YAGO is a knowledge base derived from Wikipedia, WordNet, and GeoNames. Its newest version, YAGO2s, has more than 10M entities and 120M facts, including the schema, taxonomy, core facts, etc. We use the schema for rule construction and the core 4.48M binary facts for rule evaluation.

Freebase. Freebase is a community-curated knowledge base of well-known people, places, and things, containing 112M entities and 2.68B facts, as of current writing. We preprocess the dataset by removing the multi-language support and use the remaining 388M facts. The datasets statistics are summarized in Table 4 (Left).

KB	Size
YAGO2	# Entities = 834,554 # Facts = 948,047
YAGO2s	# Entities = 2,137,468 # Facts = 4,484,907
Freebase	# Entities = 111,781,246 # Facts = 388,474,630

Max length	3
Max Γ-size	3M (YAGO) 10M (Freebase)
Max # of rules	1000
Functional constraint	100
Min support	0
Min confidence	0.0

Table 4: (Left) Datasets statistics. (Right) Default parameters.

Experiment setup. We conduct all experiments on a 64-core machine running on AMD Opteron processors at 1.4GHz, with 512GB RAM and 3.1TB disk space. The OP and AMIE algorithms are implemented in Spark and Java/SQL, respectively, running on Spark 1.3.0, Java 1.8, and PostgreSQL 9.2.3.

Default parameters. Unless otherwise specified or the current parameter is under evaluation, we use the default parameters in Table 4 (Right) for the experiments. We determine the parameters by trying multiple parameter combinations and comparing the performance and result rules. For Freebase experiments in Sections 7.2 to 7.4, we report the result of one class of length 3 rules.

Rule set precision. We evaluate a rule set by assessing its most confident rules, i.e., with a minimum confidence of 0.6 and supporting at least 2 facts. Under this constraint, we define the *precision* of a rule set as the percentage of rules satisfying the above threshold that we consider correct. Each rule is rated by two independent human judges. In case of a disagreement, the judges conduct a detailed discussion until a final decision is made. We sample at most 300 rules from each rule set for human inspection.

7.1 Overall Result

To evaluate performance of the OP algorithm and compare with the state-of-the-art, we run the OP and AMIE algorithms on Freebase and YAGO. As a result, OP mines 36,625 rules in 33.22 hours from Freebase, contributing the largest first-order rules repository created from public knowledge bases. We compare the detail performance metrics, including the number and precision of mined rules, and the runtime for each knowledge base, in Table 5.

In terms of efficiency and scalability, OP outperforms AMIE in all the experiments we run. For Freebase, AMIE takes more than 5 days to generate a single rule (AMIE outputs an inference rule once it has determined its quality), whereas OP finishes in 33.22 hours. For the smaller YAGO2s KB, AMIE mines 278 rules in 5 days while OP mines 312 rules in 19.40 minutes. For YAGO2, due to its small size, partitioning and parallelization have limited advantage, so OP is only 0.97 minutes faster. The quantity of mined rules is large: 36,625 first-order rules from Freebase, spanning a variety of topics: film, book, music, computer, etc, as shown in Figures 6 and 11(3). OP mines fewer rules from YAGO2 and YAGO2s because their sizes are much smaller, and their schemas are incomplete. Possible domain and range values are missing from overloaded predicates, so the rule construction algorithm generates a subset of all possible rules from the available schema. Using a more accurate schema, e.g., the Freebase schema, improves recall.

Dataset	Algorithm	# Rules	Precision	Runtime
YAGO2	OP	218	0.35	3.59 min
	AMIE	1090	0.46	4.56 min
YAGO2s	OP	312	0.35	19.40 min
	AMIE	278+	N/A	5+ d
Freebase	OP	36,625	0.60	33.22 h
	AMIE	0+	N/A	5+ d

Table 5: Overall mining result.

In terms of precision, Freebase rules achieve 0.60, outperforming YAGO rules by more than 0.1. The precision benefits from Freebase's cleaner data and schema. To illustrate, consider Rule (1) in Figure 6 with predicates "film/film/sequel" and "film/film/-country". These predicates impose precise constraints on the data: "sequel" refers to the sequel of a film, and "country" refers to the producing country of a film. Such constraints ensure that Freebase predicates contain fine-grained and accurate data instances. On the other hand, because 1) YAGO has fewer predicates, and 2) YAGO predicates are less well-defined, YAGO generates fewer rules with lower quality. Consider the "create" predicate from YAGO2s: the domain is possibly writer, musician, filmmaker, author, etc, and the range can be book, music, film, novel, etc. Thus, "create" can be combined with any matching predicates to form candidate rules, leading to spurious results. The rule "created(x, z), created$(y, z) \rightarrow$ isMarriedTo(x, y)" illustrates this situation. Other predicates, like "playsFor", "owns", "influences", are similarly misused.

Figures 7(a)(b) report OP's performance for one class of rules of each length from 2 to 5. We mine 1,006 and 83,163 for YAGO2s (lengths 4 and 5) and Freebase (length 4) in 8.62 and 82.77 hours, respectively. For rule construction, because the schema is much smaller than the knowledge base, it takes 13.64 and 0.94 minutes for YAGO2s and Freebase, accounting for less than 3% of the runtime. The construction process for YAGO2s is slower than for Freebase since its predicates are heavily overloaded, as we discuss above, resulting in expensive joins, especially for long rules.

Analyzing the rules, we observe that 90.3% of them reduce to lengths 2 and 3 rules, as shown in Figure 7(c). We classify these rules as trivial extensions and composite rules. We call a rule *trivial extension* of another rule if it can be reduced to the other rule by applying and removing any valid rules from its body. In Figure 6, Rule (4) is a trivial extension of Rule (3) since the rule book/book/first_edition$(x, u) \rightarrow$ book/book_edition/book(u, x) infers that $v = x$ in (4). By replacing v with x and removing the applied rule, it reduces to (3). We call a rule *composite* if it can be rewritten by chaining shorter rules. For instance, Rule (5) in Figure 6 is a composite rule of (1) and (2). Rule (6) gives an example of correct and irreducible length 4 rule.

	Confidence	Rule
(1)	0.81	film/film/sequel(x, z), film/film/country$(z, y) \rightarrow$ film/film/country(x, y)
(2)	0.44	film/film/country(x, z), location/country/official_language$(z, y) \rightarrow$ film/film/language(x, y)
(3)	1.0	book/book/first_edition$(x, y) \rightarrow$ book/book/editions(x, y)
(4)	1.0	book/book/first_edition(x, u), book/book_edition/book(u, v), book/book/first_edition$(v, y) \rightarrow$ book/book/editions(x, y)
(5)	0.41	film/film/sequel(x, u), film/film/country(u, v), location/country/official_language$(v, y) \rightarrow$ film/film/language(x, y)
(6)	0.89	music/music_video/music_video_song(x, u), music/composition/recorded_as_album(u, v), music/album/artist$(v, y) \rightarrow$ music/music_video/artist(x, y)

Figure 6: Example Freebase rules.

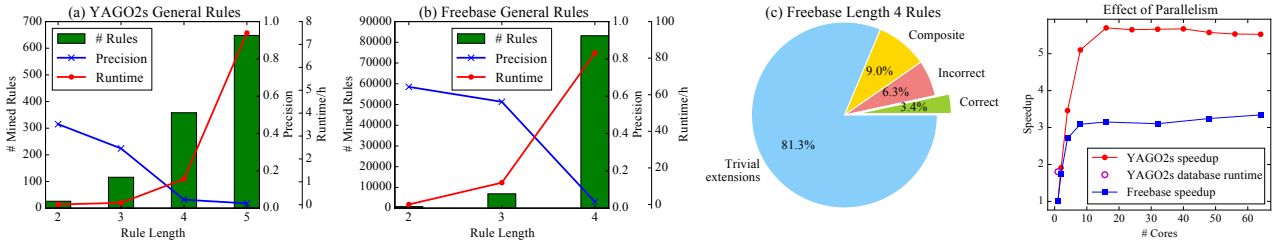

Figure 7: (a)(b) OP performance for mining lengths 4 (YAGO and Freebase) and 5 (YAGO) rules. (c) Quality of Freebase length 4 rules.

Figure 8: Effect of parallelism.

The trivial extensions and composite rules provide little knowledge in addition to lengths 2 and 3 rules. Thus, their distribution in Figure 7(c) implies limited benefits from mining longer rules. We remove those rules and evaluate the remaining rules as we do with length 2 and 3 rules. As a result, the precision is much lower than the lengths 2 and 3 rules: 0.04 for YAGO2s and 0.03 for Freebase, as reported in Figure 7. These results suggest the primary and foundational importance of shorter rules in a knowledge base. We therefore limit the maximum length to 3 in the default setting.

In summary, the overall results justify the benefits of OP algorithm to mine web-scale knowledge bases. In the remainder of this section, we examine individual techniques of parallelism, partitioning, and rule pruning in greater details and show how they improve the performance and quality of the rule mining task.

7.2 Effect of Parallelism

We evaluate the effect of parallelism by comparing the parallel mining algorithm with a SQL implementation on PostgreSQL. We vary the number of cores for parallel mining from 1 to 64 and report in Figure 8 the relative speedup compared to running on 1 core. As a result, the parallel mining algorithm achieves a speedup of 5.70 and 3.34 on YAGO2s and Freebase, respectively. Comparing with the SQL approach, parallel mining achieves a speedup of 3.16 on YAGO2s and a much higher speedup on Freebase–the SQL queries do not scale to Freebase in the experiments on PostgreSQL and on an in-house parallel database system, Datapath [3]. The speedup results from 1) the SQL query performs one large join while Algorithm 3 runs smaller joins in parallel; 2) the shuffling step in Spark is more efficient than the deduplication operation in PostgresSQL given a large output from previous joins.

These results attest to the overall advantage of parallelizing the rule mining algorithm. Nonetheless, we see the parallelization does not make full use of available 64 processors, because the output sizes and performance of GROUP-JOINs vary greatly among groups, depending on the data distribution, and is dominated by the slowest joins among the groups. Moreover, the efficiency of the shuffling step is restricted by data dependency among parallel workers.

7.3 Effect of Partitioning

Partitioning is a key step in scaling up the mining algorithm. By setting a maximum Γ-size s and number of rules m, the partitioning algorithm breaks the input knowledge base into parts with no larger than the specified size. Our experiments show the OP algorithm completes the Freebase mining task in 1.39 days with partitioning, which otherwise takes more than 5 days with no success.

The result of Freebase partitioning is illustrated in Figure 9: in the left figure, we set $s = 20$M and $m = 2$K; in the right figure, we set $s = 200$M and $m = 10$K. For the former case, we have 65 partitions, all running faster than partitions from the latter case, with the fastest partition finishing in 14.18 seconds and the slowest in 1.17 hours. For the latter case, we have 5 large partitions, the fastest taking 4.58 hours to finish and the slowest taking 1.27 days.

The effect of choosing different partition sizes is shown in Figures 10(a)-(c) for Freebase and Figure 10(d) for YAGO2s. In the Freebase experiments, the effect of partitioning is substantial: as we vary s from 200M to 5M and m from 10K to 1K, the total runtime decreases from 2.55 days to 5.06 hours. The reason for such speedup is because the partitioning algorithm manages to split the input knowledge base into smaller ones that are more efficiently joined, and the overhead of the overlap is less significant as the benefit from joining smaller tables. This benefit can be further verified by the decline of runtime of the largest partitions from 1.27 days to 38.14 minutes as we lower the size constraints, as shown in Figure 10(b), indicating that the partitions are more efficiently joined because of their smaller sizes. Consequently, the overall runtime significantly drops despite partition overlaps.

In Figure 10(c), we show the DOV increases from 1.14 to 1.35 as we create smaller partitions. The increasing DOV means we spend more time partitioning Freebase: from 2.45 minutes to 61.43 minutes. Comparing with Figure 10(a), we see the reduction from joining smaller partitions has greater impact on the total runtime.

On the other hand, if s and m become too small, the overhead of overlapping partitions begins to dominate. The overlapping effect is shown in Figure 10(d) as we partition the 4.48M YAGO2s into smaller parts: while we improve the runtime of the slowest partition from 5.80 minutes to 1.79 minutes, the total runtime rises to 29.60 minutes after hitting the optimum of 19.40 minutes when $s = 3$M and $m = 500$. The fall of performance is caused by the growing number of overlapping partitions. The extreme case of applying one rule at a time, taken by state-of-the-art approaches, is equivalent to having one rule in each partition. Given a large search space of candidate rules, it implies a large number of queries, hence too much overlapping overhead for it to be efficient.

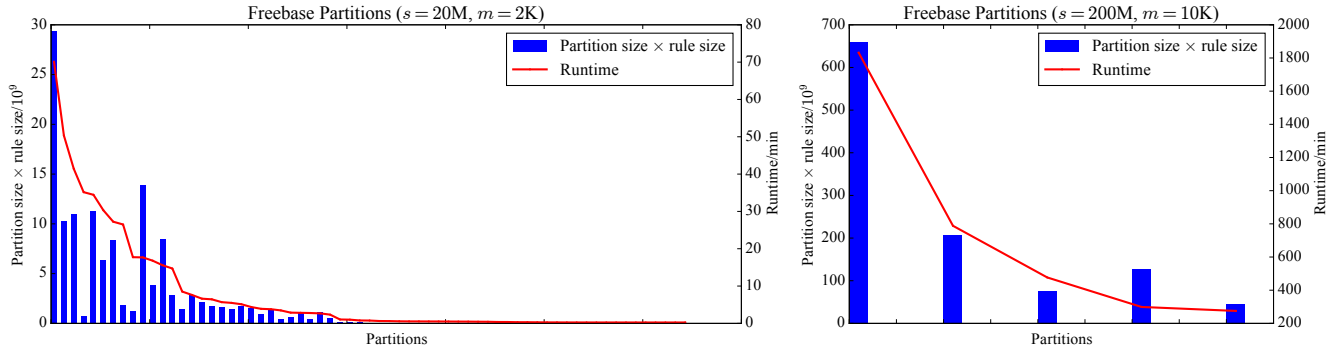

Figure 9: Sizes and runtime of Freebase partitions.

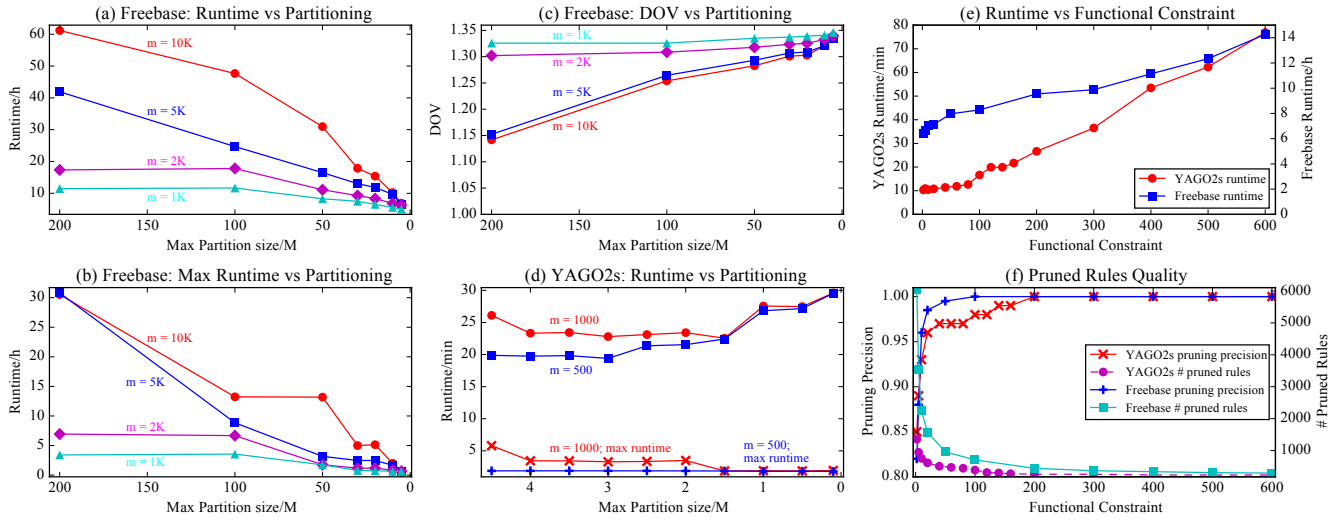

Figure 10: (a)-(d) Runtime by varying partition sizes. (e)-(f) Runtime and accuracy by varying functional constraints.

7.4 Effect of Rule Pruning

The functional constraint t affects the mining algorithm in terms of both performance and quality. To evaluate the accuracy, we define the *pruning precision* of pruned rules as the percentage of those rules that we consider erroneous. Thus, a high pruning precision indicates that erroneous rules are pruned as desired and justifies the proposed approach. As we vary the functional constraint t and prune violating rules, we report the runtime, number and pruning precision of pruned rules in Figures 10(e)(f).

We make two observations from Figures 10(e)(f): (1) When $t \geq 200$, the pruning precision reaches its maximum value of 1.0–all pruned rules are erroneous. However, the runtime grows from 9.55 hours to 14.27 hours, indicating wasted computation in evaluating wrong rules that should otherwise be eliminated by setting a smaller t. (2) On the other hand, decreasing t from 50 to 2 causes the pruning precision to drop sharply from 0.995 to 0.82 while improving runtime only by 1.54 hours from 7.97 to 6.43 hours.

From the above observations, we see the rule pruning process improves both performance and quality provided we choose a proper t constraint. Based on Figure 10(f), a value between 50 and 250 is reasonable. In our default setting, we have $t = 100$ and detect 101 and 2352 non-functional rules from YAGO2s and Freebase, respectively. For Freebase, more than 99% rules are correctly pruned. Rules (1) and (2) from Freebase in Figure 11 illustrate the common reason why functional constraints are violated: in Freebase and other knowledge bases, we have many-one, many-few, and many-many predicates. The "location/location/containedBy" predicate in Rule (1), for example, is a many-few predicate. The rule construction algorithm, based on the KB schema, is unaware of the functionality properties of the predicates. When the *one* or *few* variable happens to be the join variable, the rule violates the functional constraint. Rule (3) is an incorrectly pruned rule due to a low $t = 5$ functional constraint. The "computer/computer_processor/variants" predicate defines an equivalence relation: variants of a computer processor are variants of each other. According to the sparsity of natural graphs [21], we reduce pruning errors by raising the functional constraint t.

8. CONCLUSION

In this paper, we address the scalable first-order rule mining problem. We present the Ontological Pathfinding algorithm to mine first-order inference rules from web-scale knowledge bases. We achieve the Freebase scale via a series of parallelization and optimization techniques: a relational knowledge base model that applies inference rules in batches, a rule mining algorithm that parallelizes the join queries, a novel partitioning algorithm that divides the mining task into smaller independent sub-tasks, and a rule pruning strategy to detect incorrect and resource-consuming rules. Combining these techniques, we mine the first rule set for Freebase, the largest public knowledge base with 388 million facts and 112 million entities, in 34 hours. No existing system achieves this scale. We publish our code and data repositories online.

	Rule
(1)	location/location/containedby(x, z), location/location/contains$(z, y) \rightarrow$ location/location/contains_major_portion_of(x, y)
(2)	engineering/engine_category/engines(z, x), engineering/engine_category/engines$(z, y) \rightarrow$ engineering/engine/variants(x, y)
(3)	computer/computer_processor/variants(x, z), computer/computer_processor/variants$(y, z) \rightarrow$ computer/computer_processor/variants(x, y)

Figure 11: Example rules violating functional constraints.

Acknowledgements. This work is partially supported by NSF IIS Award # 1526753, DARPA under FA8750-12-2-0348-2 (DEFT/CUBISM), and a generous gift from Google. We also thank Dr. Milenko Petrovic and Dr. Alin Dobra for the helpful discussions on query optimization.

9. REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, 1993.

[2] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *Proceedings of the VLDB Endowment*, 1994.

[3] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*. ACM, 2010.

[4] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. *Dbpedia: A nucleus for a web of open data*. Springer, 2007.

[5] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction for the web. In *IJCAI*, 2007.

[6] J. Biega, E. Kuzey, and F. M. Suchanek. Inside yago2s: a transparent information extraction architecture. In *WWW*, 2013.

[7] G. O. Blog. Introducing the knowledge graph: thing, not strings. http://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html.

[8] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*. ACM, 2008.

[9] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010.

[10] A. Carlson, J. Betteridge, R. C. Wang, E. R. Hruschka Jr, and T. M. Mitchell. Coupled semi-supervised learning for information extraction. In *Proceedings of WSCM*, 2010.

[11] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[12] Y. Chen and D. Z. Wang. Knowledge expansion over probabilistic knowledge bases. In *SIGMOD Conference*, pages 649–660, 2014.

[13] Y. Cheng, C. Qin, and F. Rusu. Glade: big data analytics made easy. In *SIGMOD*, 2012.

[14] P. Clark, J. Thompson, and B. Porter. A knowledge-based approach to question-answering. In *In the AAAI Fall Symposium on Question-Answering Systems, AAAI*, 1999.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *SIGKDD*, 2014.

[17] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 2014.

[18] O. Etzioni, A. Fader, J. Christensen, S. Soderland, and M. Mausam. Open information extraction: The second generation. In *IJCAI*, 2011.

[19] A. Fader, S. Soderland, and O. Etzioni. Identifying relations for open information extraction. In *EMNLP*, 2011.

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

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[22] J. Han and J. Pei. Mining frequent patterns by pattern-growth: methodology and implications. *ACM SIGKDD explorations newsletter*, 2000.

[23] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *VLDB*, 2012.

[24] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: a spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013.

[25] A. Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 1951.

[26] T. N. Huynh. Discriminative learning with markov logic networks. Technical report, DTIC Document, 2009.

[27] S. Kok. *Structure Learning in Markov Logic Networks*. PhD thesis, University of Washington, 2010.

[28] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Data Mining, 2001. ICDM 2001*. IEEE, 2001.

[29] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph*. *Data mining and knowledge discovery*, 2005.

[30] N. Lao, T. Mitchell, and W. W. Cohen. Random walk inference and learning in a large scale knowledge base. In *Proceedings of EMNLP*, 2011.

[31] K. Li, D. Z. Wang, A. Dobra, and C. Dudley. Uda-gist: an in-database framework to unify data-parallel and state-parallel analytics. *Proceedings of the VLDB Endowment*, 2015.

[32] T. Lin, O. Etzioni, et al. Identifying functional relations in web text. In *EMNLP*, 2010.

[33] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, July 2010.

[35] F. Mahdisoltani, J. Biega, and F. Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *CIDR*, 2015.

[36] S. Muggleton. Inductive logic programming: derivations, successes and shortcomings. *ACM SIGART Bulletin*, 1994.

[37] S. Muggleton. Inverse entailment and progol. *New generation computing*, 1995.

[38] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *VLDB*, 2011.

[39] F. Niu, C. Zhang, C. Ré, and J. W. Shavlik. Deepdive: Web-scale knowledge-base construction using statistical learning and inference. In *VLDS*, pages 25–28, 2012.

[40] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. *SIGMOD Record*, 1995.

[41] J. R. Quinlan. Learning logical definitions from relations. *Machine learning*, 5(3):239–266, 1990.

[42] B. L. Richards and R. J. Mooney. Learning relations by pathfinding. In *Proc. of AAAI-92*, 1992.

[43] M. Richardson and P. Domingos. Markov logic networks. *Machine learning*, 62(1-2):107–136, 2006.

[44] A. Ritter, D. Downey, S. Soderland, and O. Etzioni. It's a contradiction—no, it's not: a case study using functional relations. In *EMNLP*, 2008.

[45] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the VLDB Endowment*, 1995.

[46] S. Schoenmackers, O. Etzioni, and D. S. Weld. Scaling textual inference to the web. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2008.

[47] S. Schoenmackers, O. Etzioni, D. S. Weld, and J. Davis. Learning first-order horn clauses from web text. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, 2010.

[48] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *Proceedings of the VLDB Endowment*, 2015.

[49] B. Tausend. Representing biases for inductive logic programming. In *Machine Learning: ECML-94*. Springer, 1994.

[50] D. Z. Wang, Y. Chen, C. Grant, and K. Li. Efficient in-database analytics with graphical models. *IEEE Data Engineering Bulletin*, 2014.

[51] D. Z. Wang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick. Hybrid in-database inference for declarative information extraction. In *SIGMOD*, 2011.

[52] D. Wijaya, P. P. Talukdar, and T. Mitchell. Pidgin: ontology alignment using web text as interlingua. In *CIKM*, 2013.

[53] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *SIGMOD*. ACM, 2012.

[54] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2012.

[55] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[56] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. *Proceedings of VLDB*, 2009.