Discovery of Positive and Negative Rules in Knowledge-Bases

us

ABSTRACT

We present RuDiK, a system for the discovery of declarative rules over knowledge-bases (KBs). RuDiK output is not limited to rules that rely on "positive" relationships between entities, such as "if two persons have the same parent, they are siblings", as in traditional constraint mining for KBs. On the contrary, it discovers also negative rules, i.e., patterns that lead to contradictions in the data, such as "if two person are married, one cannot be the child of the other". While the former class is fundamental to infer new relationships in the KB, the latter class is crucial for other tasks, such as error detection in data cleaning, or the creation of negative examples to bootstrap learning algorithms. The algorithm to discover positive and negative rules is designed with three main requirements: *(i)* enlarge the *expressive power* of the rule language to obtain complex rules and wide coverage of the facts in the KB, *(ii)* allow the discovery of *approximate* rules to be robust to errors and incompleteness in the KB, *(iii)* use disk-based algorithms, effectively enabling the mining of large KBs in commodity machines. We have conducted extensive experiments using real-world KBs to show that RuDiK outperform previous proposals in terms of efficiency and that it discovers more effective rules for the application at hand.

1. INTRODUCTION

Building large RDF knowledge-bases (KBs) is a popular trend in information extraction. KBs store information in the form of triples, where a *predicate*, or relation, expresses a binary relation between a *subject* and a *object*. KB triples, often referred as facts, store information about real-world entities and their relationships, such as "Michelle Obama is married to Barack Obama", or "Larry Page is the founder of Google". Significant effort has been put on KBs creation in the last 10 years in the research community (DBPedia [8], FreeBase [9], Wikidata [36], DeepDive [32], Yago [34], NELL [10], TextRunner [6]) as well as in the industry (e.g., Google [17], Wal-Mart [16]).

Unfortunately, due to their creation process, KBs are usually erroneous and incomplete. KBs are bootstrapped by extracting information from sources, oftentimes from the Web, with minimal or no human intervention. This leads to two main problems. First, errors are propagated from the sources, or introduced by the extractors, leading to false facts in the KB. Second, usually KBs do not limit the information of interest with a schema that clearly defines instance data. The set of predicates is unknown a-priori, and adding facts defined on new predicates is just a matter of inserting new triples in the KB without any integrity check. Since *closed world assumption* (CWA) does not hold [17, 22], we cannot assume that a missing fact is false, but rather we should label it as *unknown* (*open world assumption*).

As a direct consequence, the amount of errors and incompleteness in KBs is significantly larger than classic databases [35]. Since KBs are large, e.g., WIKIDATA has more than 1B facts and 300M different entities, checking all triples to find errors or add new facts cannot be done manually. A natural approach to assist curators of KBs is to discover *declarative rules* that can be executed over the KBs to improve the quality of the data [2, 11, 22]. However, these approaches so far have focused on the discovery of rules to derive new facts only, while for the first time we target the discovery of two different types of rules: *(i) positive rules*, used to enrich the KB with new facts and thus increase its coverage of the reality; *(ii) negative rules*, used to spot logical inconsistencies and identify erroneous triples.

Example 1: Consider a KB with information about parent and child relationships. A positive rule r_1 can be:

$$\texttt{parent}(b,a) \Rightarrow \texttt{child}(a,b)$$

which states that if a person a is parent of person b, then b is child of a. A negative rule r_2 has similar form, but different semantics:

$$\texttt{birthDate}(a,v_0) \wedge \texttt{birthDate}(b,v_1) \wedge v_0 > v_1 \wedge \texttt{child}(a,b) \Rightarrow false$$

The above rule states that a person b cannot be child of a if a was born after b. By instantiating the above rule for the `child` facts in the KB, we may discover erroneous triples stating that a child is born before a parent.

While intuitive to humans, the rules above must be manually stated in a KB in order to be enforced, and there are thousands of rules in a large KB with hundreds of predicates [23]. Other than enriching and cleaning KBs, negative rules support other use cases. We will show how negative rules can improve Machine Learning tasks by providing meaningful training examples [30, 32].

Proceedings of the VLDB Endowment, Vol. 9, No. 12

However, three main challenges arise when discovering rules for KBs.

Errors. Traditional database techniques for rules discovery make the assumption that data is either clean or has a negligible amount of errors [4, 12, 25, 37]. We will show that KBs present a high percentage of errors and we need techniques that are noise tolerant.

Open World Assumption. KBs contain only positive statements, and, without CWA, we cannot derive negative statements as counter examples – classic databases approaches rely on the presence of positive and negative examples [15, 29] or a fixed schema given as input.

Volume. Existing approaches for discovery of positive rules in KBs assume that data can fit into memory, something that it is not realistic given the large, and increasing, size of the KB [2, 22]. More recent approaches also rely on main memory algorithm and try to solve this problem with distributed architectures [11, 21].

We advocate that a rule discovery system should be designed to discover *approximate rules* since errors and incompleteness are in the nature of the KBs. Also, the main memory constraint not only reduces the applicability of the system, but also forces limitations on the *expressive power* of the language in order to reduce the search space. In fact, the larger is the number of patterns that can be expressed in the rules, the larger is the number of new facts and errors that can be identified, and with increasing precision. However, this comes with a computational cost, as the search space quickly becomes much larger. For this reason, existing approaches for mining positive rules prune aggressively the search space, and rely on a simple language [2, 11, 22].

We present RuDiK (Rules Discovery in Knowledge Bases), a novel system for the discovery of positive and negative rules over KBs that addresses the challenges above. RuDiK is the first system capable of discovering both approximate positive and negative rules without assuming that the KB fits into memory by exploiting the following contributions.

Problem Definition. We formally define the problem of rules discovery over erroneous and incomplete KBs. The input of the problem consists of a target predicate, from which we identify sets of positive and negative examples. In contrast with traditional ranking of rules based on a measure of support [15, 22, 28, 31], our problem definition aims at the identification of a subset of approximate rules. Given errors in the data and incompleteness, the ideal solution is a compact set of rules that cover the majority of the positive examples, and as few negative examples as possible. We map the problem to the well-known weighted set cover problem.

Examples Generation. Positive and negative examples for a predicate are crucial for our approach and determine the ultimate quality of the rules. However, crafting a large number of examples is a tedious exercise that requires manual work. Moreover, to effectively steer the algorithm towards useful rules, the examples must have properties, such as the existence of at least a meaningful path between pairs of entities in each example. We design an algorithm for examples generation that is aware of the errors and inconsistency in the KB. Our generated examples lead to better rules than examples obtained with baseline approaches.

Search Algorithm. We give a `log(k)` approximation of the problem close in spirit to the greedy algorithm for set cover, where k is the maximum number of positive examples covered by a single rule. We discover the rules to feed the algorithm by judiciously using the memory. The algorithm incrementally materializes the KB as a directed graph, and discovers rules by navigating valid paths. To reduce the use of resources, at each iteration we follow the most promising path in a A^* traversal fashion, allowing the pruning of unpromising paths. By materializing only the portion of the KB that is needed for the discovery, and generating nodes and edges *on demand*, the disk is accessed only whenever the navigation of a specific path is required. The significant reduction of the search space reduces the running time an order of magnitude in the best case scenario.

We experimentally verify the performance of rules discovery in RuDiK using real-world datasets (Section 5). The evaluation is carried on the 3 most popular and widely used KBs, namely DBPEDIA [8], YAGO [33], and WIKIDATA [36]. We show that our problem formulation leads to approximate rules that are better in quality than previous systems and that the search algorithm delivers accurate rules both in the positive and negative settings, clearly outperforming state-of-the-art systems. We also demonstrate how negative rules have a beneficial impact when providing representative examples for Machine Learning algorithms.

2. PRELIMINARIES AND DEFINITIONS

We focus on discovering rules from RDF KBs. An RDF KB is a database that represents information through RDF triples $\langle s, p, o\rangle$, where a *subject* is connected to a *object* via a *predicate*. Triples are often called *facts*. For example, the fact that Scott Eastwood is the child of Clint Eastwood could be represented with the triple $\langle Clint\_Eastwood, child, Scott\_Eastwood\rangle$. RDF KB triples respect three constraints: (i) triple subjects are always *entities*, i.e., concepts from the real world; (ii) triple objects can be either entities or *literals*, i.e., primitive types such as numbers, dates, and strings; (iii) triple predicates specifies real-world relationships between subjects and objects.

Differently from relational databases, KBs do not have a schema that defines allowed instance data. The set of predicates is unknown a-priori, and new predicates are added by inserting new triples. This model allows great flexibility, but the likelihood of introducing errors is higher than traditional schema-guided databases. While KBs can include *T-Box* facts in order to define classes, domain/co-domain types for predicates, and relationships among classes to check integrity and consistency, in most KBs – including the ones we use in our experiments – such information is missing. Hence our focus on the *A-Box* facts that describe instance data.

2.1 Language

Our goal is to automatically discover first-order logical formulas in KBs. More specifically, we target the discovery of *Horn Rules*. A Horn Rule is a disjunction of *atoms* with at most one unnegated atom. When written in the implication form, Horn Rules have one of the following format:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow B \qquad\qquad A_1 \wedge A_2 \wedge \cdots \wedge A_n \Rightarrow false$$

where $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ consists of the *body* of the rule (a conjunction of atoms), while B is the *head* of the rule (a single atom). The head of the rule is either unnegated (left) or empty (right). We call the former *definite clause* or simply *positive rule*, as it generates new positive facts, while the latter *goal clause* or *negative rule*, as it identifies false statements. In a KB, an atom is a predicate connecting two variables, two entities, or an entity and a variable. For simplicity, we represent an atom with the notation $\texttt{rel}(a,b)$, where $\texttt{rel}$ is a predicate, and a and b are either variables or entities. Given a Horn Rule r, we define r_{body} and r_{head} as the body and the head of the rule. We define the variables appearing in the head of the rule as the *target variables*. For the sake of presentation, we will also write negative rules as definite clauses by rewriting a body atom in its negated form in the head. The result is a logically equivalent formula that emphasises the generation of negative facts.

Example 2: Rule r_1 in Example 1 is a traditional positive rule, where new positive facts are identified with target variables a and b. Rule r_2 is a negative rule to identify errors, as in denial constraints for relational data [12]. However, for other applications, we can rewrite it as a definite clause to derive false facts from the KB and obtain r'_2:

$$\texttt{birthDate}(a,v_0) \wedge \texttt{birthDate}(b,v_1) \wedge v_0 > v_1 \Rightarrow \neg\texttt{child}(a,b)$$

As shown in the example, we allow *literals comparisons* in our rules. A literal comparison is a special atom $\texttt{rel}(a,b)$, where $\texttt{rel} \in \{<, \leqslant, \neq, >, \geqslant\}$, and a and b can only be assigned to literal values except if $\texttt{rel}$ is equal to $\neq$. In such a case a and b can be also assigned to entities. We will explain later on why this exception is important.

Given a KB kb and an atom $A = \texttt{rel}(a,b)$ where a and b are two entities, we say that A *holds* over kb iff $\langle a, \texttt{rel}, b\rangle \in kb$. Given a KB kb and an atom $A = \texttt{rel}(a,b)$ with at least one variable, we say that A can be *instantiated* over kb if there exists at least one entity from kb for each variable in A such that if we substitute variables with entities in A, A holds over kb. Transitively, given a body of a rule r_{body} and a KB kb, we say that r_{body} can be instantiated over kb if every atom in r_{body} can be instantiated.

Following the biases introduced by other approaches for rules discovery in KBs [11,22], we adopt also two constraints on the variables in the rules. We define a rule *valid* iff it satisfies the following constraints.

Connectivity. An atom A_1 can be reached by an atom A_2 iff A_1 and A_2 share at least one variable or one entity. The connectivity constraint requires that every atom in a rule must be *transitively* reached by any other atom in the rule.

Repetition. Every variable in a rule must appear at least twice. Since target variables already appear once in the head of the rule, the repetition constraint limited to the body of a rule requires that each variable that is not a target variable must be involved in a join or in a literal comparison.

Language restrictions limit the output of the discovery algorithm to a subset of plausible rules. We will show in Section 3 how these restrictions enable us to speed up the discovery process.

2.2 Rules Coverage

Given a pair of entities (x,y) from a KB kb and a Horn Rule r, we say that r_{body} *covers* (x,y) if $(x,y) \models r_{body}$. In other words, given a Horn Rule $r = r_{body} \Rightarrow \texttt{r}(a,b)$, r_{body} covers a pair of entities $(x,y) \in kb$ iff we can substitute a with x, b with y, and the rest of the body can be instantiated over kb. Given a set of pair of entities $E = \{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$ and a rule r, we denote by $C_r(E)$ the *coverage* of r_{body} over E as the set of elements in E covered by r_{body}: $C_r(E) = \{(x,y) \in E | (x,y) \models r_{body}\}$.

Given the body r_{body} of a Horn Rule r, we denote by r^*_{body} the *unbounded body* of r. The unbounded body of a rule is obtained by keeping only atoms that contain a target variable and substituting such atoms with new atoms where the target variable is paired with a new unique variable. As an example, given $r_{body} = \texttt{rel}_1(a,v_0) \wedge \texttt{rel}_2(v_0,b)$ where a and b are the target variables, $r^*_{body} = \texttt{rel}_1(a,v_1) \wedge \texttt{rel}_2(v_2,b)$. While in r_{body} the target variables are bounded to be connected by variable v_0, in r^*_{body}, the target variables are not bounded. Given a set of pair of entities $E = \{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$ and a rule r, we denote by $U_r(E)$ the *unbounded coverage* of r^*_{body} over E as the set of elements in E covered by r^*_{body}: $U_r(E) = \{(x,y) \in E | (x,y) \models r^*_{body}\}$. Note that, given a set E, $C_r(E) \subseteq U_r(E)$.

Example 3: Given rule r'_2 of Example 2 and a KB kb, we denote with E the set of all possible pairs of entities in kb. The coverage of r'_2 over E ($C_r(E)$) is the set of all pairs of entities $(x,y) \in kb$ s.t. both x and y have the $\texttt{birthDate}$ information and x is born after y. The unbounded coverage of r over E ($U_r(E)$) is the set of all pairs of entities (x,y) s.t. both x and y have the $\texttt{birthDate}$ information, no matter what the relation between the two birth dates is.

The unbounded coverage is essential to distinguish between missing and inconsistent information: if for a pair of entities (x,y) the $\texttt{birthDate}$ information is missing for either x or y (or both), we cannot say whether x was born before or after y. But if both x and y have the $\texttt{birthDate}$ information and x is born before y, we can affirm that r'_2 does not cover (x,y). Given that KBs are largely incomplete [17,27], discriminating between missing and conflicting information is of paramount importance.

We can now define the coverage and the unbounded coverage for a set of rules $R = \{r_1, r_2, \cdots, r_n\}$ as the union of individual coverages:

$$C_R(E) = \bigcup_{r \in R} C_r(E) \qquad U_R(E) = \bigcup_{r \in R} U_r(E)$$

Our problem tackles the discovery of rules for a *target predicate* given as input. We uniquely identify a predicate with two different sets of pairs of entities. G – *generation set*: G contains examples for the target predicate, e.g., G contains examples of parents and children if we are discovering positive rules for a child predicate. V – *validation set*: V contains counter examples for the target predicate, e.g., pairs of people that *are not* in a child relation. We will explain in Section 3.2 how to generate these two sets for a given predicate. Note that our approach is not less generic than those for mining rules for an entire KB (e.g., [2,22]): we can apply our setting for every predicate in the KB and compute rules for each of them (see Section 5.2).

We can now formalise the *exact discovery problem*.

Definition 1: Given a KB kb, two sets of pairs of entities G and V from kb such that $G \cap V = \varnothing$, and a universe of Horn Rules R, a solution for the *exact discovery problem* is a subset R' of R such that:

$$R_{opt} = \underset{|R'|}{\operatorname{argmin}}(R' | (C_{R'}(G) = G) \wedge (C_{R'}(V) \cap V = \varnothing))$$

The exact solution is a set of rules that covers all examples in G, and none of the examples in V.

Example 4: Consider the discovery of positive rules for the predicate `couple` between two persons using as examples the Obama family. A positive example is (Michelle, Barack) and a negative example their daughters (Malia, Natasha). Given two positive rules:

$$\texttt{livesIn}(a, v_0) \wedge \texttt{livesIn}(b, v_0) \Rightarrow \texttt{couple}(a, b)$$
$$\texttt{hasChild}(a, v_1) \wedge \texttt{hasChild}(b, v_1) \Rightarrow \texttt{couple}(a, b)$$

The first rule states that two persons are a couple if the live in the same place, while the second states that they are a couple if they have a child in common. Both rules cover the positive example, but only the second rule does not cover the negative one as all of them live in the same place.

In the problem definition, the solution minimizes the number of rules in the output to avoid precise rules covering only one pair, which are not useful when applied on the entire KB. Note in fact that given a pair of entities (x, y), there is always a Horn Rule whose body covers only (x, y) by assigning target variables to x and y (e.g., $\texttt{hasChild}(Michelle, v_1) \wedge \texttt{hasChild}(Barack, v_1) \Rightarrow \texttt{couple}(Michelle, Barack)$).

Unfortunately this definition leads to poor rules because of the data problems in KBs. Even if a valid, general rule exists semantically, missing information or errors for the examples in G and V can lead to faulty coverage, e.g., the rule misses a good example because a child relation is missing for M. Obama. The exact solution may therefore be a set of rules where every rule covers only one example in G and none in V, ultimately leading to a set of overfitting rules.

2.3 Weight Function

Given the errors and missing information in both G and V, we drop the strict requirement of the coverage of G and V. However, coverage is a strong indicator of quality: good rules should cover several examples in G, while covering several elements in V is an indication of potentially incorrect rules, as we assume that errors are always a minority in the data. We therefore define a *weight* to be associated with a rule.

Definition 2: Given a KB *kb*, two sets of pair of entities G and V from *kb* such that $G \cap V = \emptyset$, and a Horn Rule r, the weight of r is defined as follow:

$$w(r) = \alpha \cdot \left(1 - \frac{|\, C_r(G) \,|}{|\, G \,|}\right) + \beta \cdot \left(\frac{|\, C_r(V) \,|}{|\, U_r(V) \,|}\right) \qquad (1)$$

with $\alpha, \beta \in [0, 1]$ and $\alpha + \beta = 1$.

The weight is a value between 0 and 1 that captures the quality of a rule w.r.t. G and V: the better the rule, the lower the weight – a perfect rule covering all elements of G and none of V would have a weight of 0. The weight is made of two components normalized by the two parameters α and β. *1)* The first component captures the coverage over the generation set G – the ratio between the coverage of r over G and G itself. *2)* The second component aims at quantifying potential errors of r by using the coverage over V. The coverage over V is not divided by total elements in V because some elements in V might not have predicates stated in r_{body}. Thus we divide the coverage over V by the unbounded coverage of r over V.

Parameters α and β give relevance to each component. A high β steers the discovery towards rules with high precision by penalizing the ones that cover negative examples, while a a high α champions the recall as the discovered rules identify as many examples as possible.

Example 5: Consider again the negative rule r'_2 of Example 2 and two sets of pairs of entities G and V from a KB *kb*. The two components of w_r are computed as follow: *1)* the first component is computed as 1 minus the number of pairs (x, y) in G where x is born after y divided by the total number of elements in G; *2)* the second component is the ratio between number of pairs (x, y) in V where x is born after y and number of pairs (x, y) in V where the date of birth (for both x and y) is available in *kb*.

Definition 3: Given a set of rules R, the weight for R is defined as:

$$w(R) = \alpha \cdot \left(1 - \frac{|\, C_R(G) \,|}{|\, G \,|}\right) + \beta \cdot \left(\frac{|\, C_R(V) \,|}{|\, U_R(V) \,|}\right)$$

Weights enable the modeling of the presence of errors in KBs. We will show in the experimental evaluation that several semantically correct rules have a significant coverage over V, which corresponds to errors in the KB.

2.4 Problem Definition

We can now state the approximate version of the problem.

Definition 4: Given a KB *kb*, two sets of pair of entities G and V from *kb* where $G \cap V = \emptyset$, a universe of rules R, and a w weight function for R, a solution for the *approximate discovery problem* is a subset R' of R such that:

$$R_{opt} = \underset{w(R')}{\operatorname{argmin}}(R' | R'(G) = G)$$

We can map this problem to the well-known weighted set cover problem, which is proven to be NP-complete [14]. The universe corresponds to G and the input sets are all the possible rules defined in R.

Since we want to minimize the total weight of the output rules, the approximate version of the discovery problem aims to cover all elements in G, and as few as possible elements in V. Since for each element $g \in G$ there always exists a rule that covers exactly only g (single-instance rule), an optimal output is always guaranteed to exist. We expect the output to be made of some rules that cover multiple examples in G, while remaining examples in G to be covered by single-instance rules.

3. RULES AND EXAMPLES GENERATION

In this Section we introduce the techniques to create the candidate rules, assuming the examples are given, and then show how positive and negative examples can be automatically computed. However, our approach is independent on how G and V are generated: they could also be manually crafted by some domain experts, which would require additional manual effort.

In the following we focus on the discovery of positive rules, i.e., having true facts in G and false facts in V. However, in the dual problem of negative rules discovery our approach remains unchanged, we just switch the role of G and V. The generation set becomes V (negative examples), while the validation set becomes G (positive examples).

3.1 Rules Generation

The first task we address is the generation of the universe of all possible rules R. The number of possible rules increases exponentially with the size of the KB and enumerat-

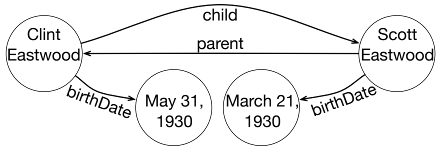

Figure 1: Graph Portion of DBPedia

ing all of them leads to exploring a huge search space. While loading the KB in memory alleviates this problem [11, 22], modern KBs can easily exceed the size of hundreds of GBs, making a memory-based solution unfeasible without significant hardware, such as a distributed platform [11, 21]. On the contrary, our rule generation technique inspects only a portion of the KB without losing any rule that is good w.r.t. our weighting scheme. Also, since we load into main memory a small fraction of the database, we can afford a more expressive language compared to previous works.

In our approach, each rule in R must cover one or more examples in the generation set G. Therefore the universe of all possible rules is generated by inspecting elements of G only. The smaller the size of G is, the smaller the search space for rule generation.

We translate a KB kb into a directed graph: entities and literals are nodes of the graph, while there is a direct edge from node a to node b for each triple $\langle a, rel, b\rangle \in kb$. Edges are labelled, where the label is the relation rel that connects subject to object in the triple. Figure 1 shows a portion of DBPedia [8] that connects two person in child and parent relationships, along with their dates of birth. The graph represents information for four KB triples.

The body of a rule can be seen as a path in the graph. In Figure 1, the body $\texttt{child}(a, b) \wedge \texttt{parent}(b, a)$ corresponds to the path *Scott Eastwood → Clint Eastwood → Scott Eastwood*. As introduced in Section 2.1, a valid body of a rule contains target variables a and b at least once, every other variable at least twice, and atoms are transitively connected. If we allow navigation of edges independently of the edge direction, we can translate bodies of valid rules to valid paths on the graph. Given a pair of entities (x, y), a *valid body* corresponds to a valid path p on the graph that: (i) p starts at the node x; (ii) p covers y at least once; (iii) p ends in x, in y, or in a different node that has been already covered.

In other words, given the body of a rule r_{body}, r_{body} covers a pair of entities (x, y) iff there exists a valid path on the graph that corresponds to r_{body}. This implies that for a pair of entities (x, y), we can generate bodies of all possible valid rules by computing all valid paths from x to y (and vice versa) with a standard BFS. The key point is the ability of navigating each edge in any direction by turning the original directed graph into an undirected one. However, we need to keep track of the original direction of the edges. This is essential when translating paths to rule bodies. In fact, navigating an edge from a to b produces the atom $\texttt{rel}(a, b)$, while b to a produces $\texttt{rel}(b, a)$.

One can notice that for two entities x and y, there might exist infinite valid paths starting from x, since every node can be traversed multiple times. Thus we introduce the $maxPathLen$ parameter that determines the maximum number of edges in the path. When translating paths to Horn Rules, $maxPathLen$ determines the maximum number of atoms that we can have in the body of the rule. This parameter is essential to avoid the discovery of rules with infinite body length and to constraint the search space.

We now describe our algorithm for the generation of the universe of all possible rules for G.

Create Paths. Given a pair of entities (x, y), we retrieve from the KB all nodes at distance smaller than $maxPathLen$ from x or y, along with their edges. The retrieve is done recursively: we maintain a queue of entities, and for each entity in the queue we execute a SPARQL query against the KB to get all entities (and edges) at distance 1 from the current entity – we call these queries *single hop queries*. At the n-th step, we add the new found entities to the queue iff they are at distance less than $maxPathLen - n$ from x or y and they have not been visited before. The queue is initialized with x and y. Given the graph for every (x, y), we then compute all valid paths starting from every x.

Evaluate Paths. Computing paths for every example in G implies also computing the coverage over G for each rule. The *coverage* of a rule r is the number of elements in G for which there exists a path corresponding to r_{body}. Our technique also generates single-instance rules: rules that cover only one example $(x, y) \in G$ by instantiating target variables a and b in the rule with x and y. Once the universe of all possible rules has been generated (along with coverages over G), computing coverage and unbounded coverage over V is just a matter of executing two SPARQL queries against the KB for each rule in the universe.

Generating Complex Rules

Literals comparison. In Section 2.1 we defined our target language, which, other than predicate atoms, includes literals comparison. Their scope is to enrich the language with comparisons among literal values other than equalities, such as “greater than” or “less than”. To discover such kind of atoms, the graph representation must contain edges that connect literal values with one (or more) symbol from $\{<, \leqslant, \neq, >, \geqslant\}$. As an example, Figure 1 should contain an edge ‘$<$’ from node “*March 31, 1930*” to node “*March 21, 1986*”. Unfortunately, the original KB does not contain this kind of information, and materializing such edges among all literals is unfeasible.

The generation set G is the key point to introduce literals comparison. Since we discover paths for a pair of entities from G in isolation, the size of a graph for a pair of entities is relatively small, thus we can afford to compare all literal values within a single example graph. Despite this implies the creation of a quadratic number of edges w.r.t. the number of literals in the graph, within a single example graph the number of literals is usually small thus the quadratic comparison affordable. KBs include three types of literals: numbers, dates, and strings. Besides equality comparisons, we add ‘$>$’,‘$\geqslant$’,‘$<$’,‘$\leqslant$’ relationships between numbers and dates, and $\neq$ between all literals. These new relationships are treated as normal atoms (edges): $x \geqslant y$ is equivalent to $\texttt{rel}(x, y)$, where $\texttt{rel}$ is equal to $\geqslant$.

Not equal variables. The “not equal” operator introduced for literals is useful for entities as well. For example, consider the negative rule:

$$\texttt{bornIn}(a, x) \wedge x \neq b \Rightarrow \neg\texttt{president}(a, b)$$

The rule states that if a person a is born in a country that is different from b, then a cannot be president of b. One way to consider inequalities among entities is to add edges among all pairs of entities in the graph. However, this strategy is inefficient and would lead to many meaningless rules. To limit the search space and aiming at meaningful rules, we use the `rdf:type` triples associated to entities. We therefore add an artificial inequality edge in the input example graph only between those pairs of entities of the same type (as in the president example above).

Constants. Finally, we allow the discovery of rules with constant selections. Suppose that for the above negative rule for president, all examples in G are people born in the country "*U.S.A.*" and there is at least a country for which this rule is not valid. According to our problem statement, the right rule is therefore:

$$\texttt{bornIn}(a, x) \wedge x \neq U.S.A. \Rightarrow \neg\texttt{president}(a, U.S.A.)$$

To discover such rules, we introduce a refinement of the rule generation. For a given rule r, we promote a variable v in r to an entity e iff for every $(x, y) \in G$ covered by r, v is always instantiated with the same value e.

3.2 Input Examples Generation

Given an input KB kb and a predicate $rel \in kb$, we automatically build a generation set G and a validation set V as follows. G consists of positive examples for the target predicate rel. It is generated as all pairs of entities (x, y) such that $\langle x, rel, y\rangle \in kb$. V consists of counter (negative) examples for the target predicate. These are more complicated to generate because of the open world assumption in KBs. Differently from classic database scenarios, we cannot assume that what is not stated in a KB is false (closed world assumption), thus everything that is not stated is *unknown*.This implies that we cannot take two entities that are not related by a property and assume that together they form a negative example.

To mitigate this problem, and generate negative examples that are likely to be correct (true false facts), we are after entities that are more likely to be complete in their information. For this task, we exploit the *Local-Closed World Assumption* (*LCWA*) [17, 22]. LCWA states that if a KB contains one or more object values for a given subject and predicate, then it contains all possible values. For example, if a KB contains one or more children of Clint Eastwood, then it contains all his children. This is always true for *functional* predicates (e.g., `capital`), while it might not hold for non-functional ones (e.g., `child`). KBs contain many non-functional predicates, we therefore extend the definition of LCWA by considering the dual aspect: if a KB contains one or more subject values for a given object and predicate, then it contains all values.

Now that we have entities that are likely to be complete, we could generate negative examples taking the union of entities satisfying the LCWA: for a predicate rel, a negative example is a pair (x, y) where either x is the subject of one or more triples $\langle x, rel, y'\rangle$ with $y \neq y'$, or y is the object of one or more triples $\langle x', rel, y\rangle$ with $x \neq x'$. For example, if rel = `child`, a negative example is a pair (x, y) such that x has some children in the KB that are not y, or y is the child of someone that is not x. By considering the union of the two LCWA aspects we do not restrict predicates to be functional (such as in [22]).

The number of negative examples generated with the above technique is extremely large; for a target `child` predicate it is close to the cartesian product of all the people having a child with all the people having a parent. However, to apply the same approach for positive and negative rules discovery, we require G and V to be of comparable sizes. Instead of randomly selecting a subset, we introduce a further constraint that significantly reduces the size of V. Given a candidate negative example over entities (x, y), x must be connected to y via a predicate that is different from the target predicate. In other words, given a KB kb and a target predicate rel, (x, y) is a negative examples if $\langle x, rel', y\rangle \in kb$, with $rel' \neq rel$. This intuition exploits the LCWA for predicates rather than for entities. If a KB contains a relation between two entities x and y, then it contains all relations between x and y. This further restriction has two main advantages: (i) it makes the size of V of the same order of magnitude of G (see Section 5), and (ii) it guarantees the existence of a path between x and y, for every $(x, y) \in V$. Since V becomes the generation set in the negative rules discovery, V must be small in size and it must guarantee the existence of a path between pairs of entities in each example. Such a property was already guaranteed in G, since pairs are always connected at least by the target predicate.

Example 6: A negative example (x, y) for the target predicate `child` has the following characteristics: *(i)* x and y are not connected by a `child` predicate; *(ii)* either x has one or more children (different from y) or y has one or more parents (different from x); *(iii)* x and y are connected by a predicate that is different from `child` (e.g., `colleague`).

KBs use the same predicate to connect different types. For example, the `child` predicate is used to connect two entities of type person, but also two companies in an ownership relation. To enhance the quality of the input examples and avoid cases of mixed types, we introduce the *type* restriction when generating G and V. The type restriction requires that for every example pair (x, y) belonging to either G or V, x and y are always of the same type across the pairs.

4. DISCOVERY ALGORITHM

We introduce a greedy approach to solve the approximate version of the discovery problem (Section 2.4). The algorithm combines two phases: *(i)* it solves the set cover problem with a greedy strategy; *(ii)* it discovers new rules by navigating the graph in a A^* search fashion.

4.1 Marginal Weight

We follow the intuitions behind the greedy algorithm for weighted set cover by defining a *marginal weight* for rules that are not yet included in the solution [14].

Definition 1: Given a set of rules R and a rule r such that $r \notin R$, the marginal weight of r w.r.t. R is defined as:

$$w_m(r) = w(R \cup \{r\}) - w(R)$$

The marginal weight quantifies the total weight increase of adding r to an existing set of rules R. In other words, it indicates the contribution of r to R in terms of new elements covered in G and new elements unbounded covered in V. Due to the first negative part of the weight, $w_m(r) \in [-\alpha, +\beta]$. Since the set cover problem aims at minimizing the total weight, we would never add a rule to the solution if its marginal weight is greater than or equal to 0.

The algorithm for greedy rule selection is then straightforward: given a generation set G, a validation set V, and the universe of all possible rules R, the algorithm picks at each iteration the rule r with minimum marginal weight and add it to the solution R_{opt}. The algorithm stops when one of the following termination conditions are met: *1)* R is empty – all the rules have been included in the solution; *2)* R_{opt} covers all elements of G; *3)* the minimum marginal weight is greater than or equal to 0 – among the remaining rules in R, none of them has a negative marginal weight. If the second termination condition is not met, there may exist examples in G that are not covered by R_{opt}. In such a case the algorithm will augment R_{opt} with single-instance rules (rules that cover only one example), one for each element in G not covered by R_{opt}.

Since the coverage of a rule is always contained in its unbounded coverage, the marginal weight is greater than or equal to 0 whenever the rule does not cover new elements in G and does not unbounded cover new elements in V.

The greedy solution guarantees a $\texttt{log}(k)$ approximation to the optimal solution [14], where k is the largest number of elements covered in G by a rule in R – k is at most $|G|$. If the output rules in the final solution cover disjoint sets of G, then the greedy solution coincides with the optimal one.

4.2 A^* Graph Traversal

The greedy algorithm for weighted set cover assumes that the universe of rules R has been generated. To generate R, we need to traverse all valid paths from a node x to a node y, for every pair $(x, y) \in G$. But do we really need all possible paths for every example in G?

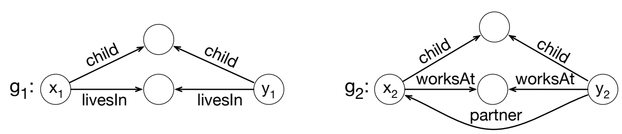

Figure 2: Two positive examples.

Example 7: Consider the scenario where we are mining positive rules for the target predicate `spouse`. The generation set G includes two examples g_1 and g_2, Figure 2 shows the corresponding KB graphs. Assume for simplicity that all rules in the universe have the same coverage and unbounded coverage over the validation set V. One candidate rule is $r : \texttt{child}(x, v_0) \wedge \texttt{child}(y, v_0) \Rightarrow \texttt{spouse}(x, y)$, stating that entities x and y with a common child are married. Looking at the KB graph, r covers both g_1 and g_2. Since all rules have the same coverage and unbounded coverage over V, there is no need to generate any other rule. In fact, any other candidate rule will not cover new elements in G, therefore their marginal weights will be greater than or equal to 0. Thus the creation and navigation of edges `livesIn` in g_1, `worksAt` in g_2, and `partner` in g_2 become worthless.

Based on the above observation, we avoid the generation of the entire universe R, but rather consider at each iteration the most promising path on the graph. The same intuition is behind the A^* graph traversal algorithm [24]. Given an input weighted graph, A^* computes the smallest cost path from a starting node s to an ending node t. At each iteration, A^* maintains a queue of partial paths starting from s, and it expands one of these paths based on an *estimation* of the cost to reach t. The path with the best estimation is expanded and added to the paths queue. The algorithm keeps iterating until one of the partial paths reaches t.

We discover rules with a similar technique. For each example $(x, y) \in G$, we start the navigation from x. We keep a queue of not valid rules, and at each iteration we consider the rule with the minimum marginal weight, which corresponds to paths in the example graphs. We expand the rule by following the edges, and we add the new founded rules to the queue of not valid rules. Differently from A^*, we do not stop when a rule (path) reaches the node y. Whenever a rule becomes valid, we add the rule to the solution and we do not expand it any further. The algorithm keeps looking for plausible paths until one of the termination conditions of the greedy cover algorithm is met.

A crucial point in A^* is the definition of the estimation cost. To guarantee the solution to be optimal, the estimation must be *admissible*, i.e., the estimated cost must be less than or equal to the actual cost. For example, for the shortest route problem, an admissible estimation is the straight-line distance to the goal for every node, as it is physically the smallest distance between any two points. In our setting, given a rule that is not yet valid and needs to be expanded, we define an admissible estimation of the marginal weight.

Definition 2: Given a rule $r : A_1 \wedge A_2 \cdots A_n \Rightarrow B$, we say that a rule r' is an *expansion* of r iff r' has the form $A_1 \wedge A_2 \cdots A_n \wedge A_{n+1} \Rightarrow B$.

In other words, expanding r means adding a new atom to the body of r. In the graph traversal, expanding r means traversing one further edge on the path defined by r_{body}. To guarantee the optimality condition, the estimated marginal weight for a rule r that is not valid must be less than or equal to the actual weight of any valid rule that is generated by expanding r. Given a rule and some expansions of it, we can derive the following.

Lemma 4.1: *Given a rule r and a set of pair of entities E, then for each r' expansion of r, $C_{r'}(E) \subseteq C_r(E)$ and $U_{r'}(E) \subseteq U_r(E)$.*

The above Lemma states that the coverage and unbounded coverage of an expansion r' of r are contained in the coverage and unbound coverage of r, respectively, and directly derives from the augmentation inference rule for functional dependencies [4].

The only positive contribution to marginal weights is given by $|C_{R\cup\{r\}}(V)|$. $|C_{R\cup\{r\}}(V)|$ is equivalent to $|C_R(V)|+|C_r(V)\backslash C_R(V)|$, thus if we set $|C_r(V)\backslash C_R(V)| = 0$ for any r that is not valid, we guarantee an admissible estimation of the marginal weight. We therefore estimate the coverage over the validation set to be 0 for any rule that can be further expanded, since expanding the rule may bring its coverage to 0.

Definition 3: Given a *not valid* rule r and a set of rules R, we define the *estimated marginal weight* of r as:

$$w^*_m(r) = -\alpha \cdot \frac{| C_r(G)\backslash C_R(G) |}{| G |} + \beta \cdot \left(\frac{| C_R(V) |}{| U_{R\cup\{r\}}(V) |} - \frac{| C_R(V) |}{| U_R(V) |} \right)$$

The estimated marginal weight for a valid rule instead is equal to the actual marginal weight defined in Equation 1. Valid rules are not considered for expansion, therefore we do not need to estimate their weights since we know the actual ones. Given Lemma 1, we can easily see that $w^*_m(r) \leqslant w^*_m(r')$, for any r' expansion of r. Thus our marginal weight estimation is admissible.

We use the concept of *frontier nodes* for a rule r ($N_f(r)$). Given a rule r, $N_f(r)$ contains the last visited nodes in the paths that correspond to r_{body} from every example graphs covered by r. As an example, given $r_{body} = \texttt{child}(x, v_0)$, $N_f(r)$ contains all the entities v_0 that are children of x, for

Algorithm 1: RuDiK Rules Discovery.

input : G – generation set
input : V – validation set
input : $maxPathLen$ – maximum rule body length
output: R_{opt} – greedy set cover solution
1 $R_{opt} \leftarrow \varnothing$;
2 $N_f \leftarrow \{x | (x, y) \in G\}$;
3 $Q_r \leftarrow$ `expandFrontiers`(N_f); `// SPARQL`
4 $r \leftarrow \underset{w_m^*(r)}{\operatorname{argmin}}(r \in Q_r)$;
5 **repeat**
6   $Q_r \leftarrow Q_r \backslash \{r\}$;
7   **if** `isValid`(r) **then**
8     $R_{opt} \leftarrow R_{opt} \cup \{r\}$;
9   **else**
    `// rules expansion`
10   **if** `length`$(r_{body}) < maxPathLen$ **then**
11       $N_f \leftarrow$ `frontiers`(r);
12       $Q_r \leftarrow Q_r \cup$ `expandFrontiers`(N_f); `// SPARQL`
13   $r \leftarrow \underset{w_m^*(r)}{\operatorname{argmin}}(r \in Q_r)$;
14 **until** $Q_r = \varnothing \vee C_{R_{opt}}(G) = G \vee w_m^*(r) \geqslant 0$;
15 **if** $C_{R_{opt}}(G) \neq G$ **then**
16   $R_{opt} \leftarrow R_{opt} \cup$ `singleInstanceRule`$(G \backslash C_{R_{opt}}(G))$;
17 **return** R_{opt}

every $(x, y) \in G$. Expanding a rule r implies navigating a single edge from any frontier node. Algorithm 1 shows the modified set cover version that includes A^*-like rules generation. The set of frontier nodes is initialized with starting nodes x, for every $(x, y) \in G$ (Line 2). The algorithm maintains a queue of rules Q_r, from which it chooses at each iteration the rule with minimum approximated weight. The function `expandFrontiers` retrieves from the KB all nodes (along with edges) at distance 1 from frontier nodes and returns the set of all rules generated by this one hop expansion. Such expansions are computed with efficient single-hop SPARQL queries. Q_r is therefore initialized with all rules of length 1 starting at x (Line 3). In the main loop, the algorithm checks if the current best rule r is valid or not. If r is valid, r is added to the output and it is not expanded (Line 8). If r is not valid, r is expanded iff the length of its body is less than $maxPathLen$ (Line 10). The termination conditions and the last part of the algorithm are the same of the previously described greedy set-cover algorithm.

The simultaneous rules generation and selection of Algorithm 1 brings multiple benefits. First, we do not generate the entire graph for every example in G. Nodes and edges are generated *on demand*, whenever the algorithm requires their navigation (Line 12). If the initial part of a path is not promising according to its estimated weight, the rest of the path will never be materialized. Rather than materializing the entire graph and then traversing it, our solution gradually materializes parts of the graph whenever they are needed for navigation (Lines 3 and 12). Second, the weight estimation leads to pruning unpromising rules. If a rule does not cover new elements in G and does not unbounded cover new elements in V, then its estimated marginal weight is 0 and will be pruned away.

5. EXPERIMENTS

We carried out an extensive experimental evaluation of our rules discovery approach. We grouped the evaluation into 4 main sub-categories: *(i)* a first set of experiments aims at demonstrating the quality of our output, both for negative and positive rules; *(ii)* a second set of experiments compares our method with state-of-the-art systems; *(iii)* in the third set of experiments we outline the applicability of rules discovery by enhancing Machine Learning algorithms; *(iv)* in the last set of experiments we sketch internal system settings and some KB properties.

Settings. We evaluated our approach over several popular KBs. We downloaded the most up-to-date core facts and loaded them into our SPARQL query engine. We experimented several SPARQL engines, and eventually we opted for OpenLink Virtuoso, as it was the fastest among all the solutions. All experiments are run on a iMac desktop with an Intel quad-core i5 at 2.80GHz with 16 GB RAM. We run Virtuoso server with its SPARQL query endpoint on the same machine, optimised for a 8 GB available RAM.

Our method needs the two input parameters α and β of Equation 1. We set $\alpha = 0.3$ and $\beta = 0.7$ for positive rules, while $\alpha = 0.4$ and $\beta = 0.6$ for negative rules. We empirically justify this choice in Section 5.4. We also set the $maxPathLen$ parameter to 3. This number represents the maximum number of atoms that we can have in the body of a rule. We will outline in Section 5.4 that increasing this parameter does not bring any benefits, as body rules longer than 3 atoms start to be very complicated and not insightful.

Evaluation Metrics. RuDiK discovers rules for a given target predicate. For each KB, we chose 5 representative predicates as follows: we first ordered predicates according to descending popularity (i.e., number of triples having that predicate), and then we picked the top 3 predicates for which we knew there existed at least one meaningful rule, and other 2 top predicates for which we did not know whether some meaningful rules existed. We repeated the procedure for each input KB, and for positive and negative rules. Despite working one predicate at time, RuDiK can also discover rules for the entire KB by listing all predicates in the KB, and discovering rules for each of them. We will show in Section 5.2 how this can be achieved.

Positive rules are very useful to enrich the KB by discovering new facts. Inspired by [22], we proceed as follows: we run the algorithm over the KB, and for each output rule we generate all new predictions that are not already in the KB (we execute the body of the rule against the KB and remove all those pairs that are already connected by the target predicate in the KB). If a rule is universally correct , we mark all its new predictions as true. If a rule is unknown, we randomly sampled 30 new predictions and manually checked them against the Web. The *precision* of a rule is then computed as the ratio of true predictions out of true and false predictions.

Negative rules are slightly more complicated to evaluate. In fact, despite KBs are usually incomplete, a large percentage of the data not stated in a KB is false. Therefore negative rules will always discover many correct negative facts. However, negative rules are a great means to discover errors in the KB, and we leverage this aspect to evaluate them. For each discovered rule, we retrieve from the KB pairs of entities for which the body of the rule can be instantiated over the KB and that are also connected by the target predicate. As an example, for the rule `child`$(a, b) \Rightarrow \neg$`spouse`(a, b), we retrieve all those pairs (a, b) such that b is `child` of a and a is `spouse` with b. We call these generated pairs of entities *potential errors*. Similarly to positive rules, whenever a rule is universally correct we mark all its potential errors as true, whereas if the rule is unknown we manually check 30 sampled potential errors. The final precision of a rule is computed as actual errors divided by potential errors.

The full suite of test results, together with all the KBs, induced rules, and annotated gold standard examples and rules is available online(**TODO: add footnote online appendix or technical report**).

5.1 Rules Discovery Accuracy

The first set of experiments aims at evaluating the accuracy of discovered rules over the 3 most popular and widely used KBs: DBPEDIA [8], YAGO [33], and WIKIDATA [36]. For each KB we downloaded the most recent version and selected core facts, facts about people, geolocations and transitive `rdf:type` facts. WIKIDATA provides only the entire dump, therefore we just eliminated from it no-english literal values. Table 1 shows the characteristics of the 3 KBs.

Table 1: Dataset characteristics.

KB	*Version*	*Size*	*#Triples*	*#Predicates*
DBPEDIA	3.7	10.056GB	68,364,605	1,424
YAGO	3.0.2	7.82GB	88,360,244	74
WIKIDATA	20160229	12.32GB	272,129,814	4,108

The size of the KB is relevant. Loading the entire KB in memory is not feasible unless we have high memory availability [11,21], or we reduce the KB by eliminating literals [22]. We propose an approach that is disk-based where only a small portion of the KB is loaded in memory, such that we can discover rules on any size KB with limited resources.

Positive Rules Discovery. We first evaluate the precision of positive rules for the top 5 predicates on the 3 KBs. The number of new induced facts varies significantly from rule to rule – a rule with literals comparison will produce a very high number of facts. In order to avoid the precision to be dominated by such rules, we first compute the precision for each rule as explained above, and then we average values over all induced rules. Table 2 reports precision values, along with predicates average running time.

Our first observation is that the more accurate is the KB, the better is the quality of induced rules. WIKIDATA contains very few errors, since it is manually curated . DBPEDIA and YAGO instead are automatically generated by extracting information from the Web, hence their quality is significantly lower. Discovering perfect positive rules is a hard task, mostly because there is no guarantee of the existence of valid negative examples. A striking example in this direction is one of the rule induced for `founder` in DBPEDIA. Our approach discovers that if a person is born in the same place where a company is founded, then the person is the founder. The rule is obviously wrong . However this rule has a very high coverage over the generation set (many companies' founders founded their company in their birth place), and a very low coverage over the validation set – among the cartesian product of all the people and companies, a very small fraction includes people and companies born and founded in the same place. Despite such hard cases, our approach is always capable of producing correct rules for those predicates for which we knew there existed some valid rules. Cases like `academicAdvisor`, `child`, and `spouse` have a precision above 95% in all of the KBs, and final precision values are brought down by few predicates where meaningful rules probably do not exist at all.

Table 2: Positive Rules Accuracy.

KB	*Avg. Running Time*	*Precision*
DBPEDIA	34min, 56sec	**63.99%**
YAGO	59min, 25sec	**62.86%**
WIKIDATA	2h, 21min, 34sec	**73.33%**

The running time is influenced by different factors. First of all the size of the KB has obviously a huge impact, as RuDiK is slower in WIKIDATA which is the biggest KB. Not only the number of triples is relevant, but also the different number of predicates. In fact the more predicates we have in the KB, the more alternative paths we observe when traversing the graph, hence a bigger search space. The second relevant aspect is the target predicate involved. We noticed that some kind of entities have a huge number of outgoing and incoming edges ("*United States*" in WIKIDATA is connected to more than 600K entities). When the generation set includes such types of entities, the navigation of the graph is slower as we need to traverse a high number of edges. Eventually the *maxPathLen* parameter also has a big say in the final running time. The longer the rule, the bigger is the search space.

Negative Rules Discovery. We evaluated negative rules as the percentage of correct errors discovered for the top 5 predicates in each KB. Table 3 shows, for each KB, the total number of potential erroneous triples discovered with negative rules, whereas the precision is computed as the percentage of actual errors among potential errors.

Negative rules generally have better accuracy than positive ones. This is mostly due to the completeness of the validation set: for negative rules the validation set is the universe of all possible counter examples stated in the KB, whereas for positive rules the validation set is a small fraction of it. WIKIDATA shows lower numbers because it does not contain as many errors as DBPEDIA and YAGO: even though discovered rules are almost correct, the percentage of actual errors identified is lower in WIKIDATA. For example, we identify the same rule that two people with same gender cannot be married both in YAGO and WIKIDATA. Such a rule has a 94% precision in YAGO, while the accuracy in WIKIDATA is 57%. YAGO is the KB with the highest number of errors. As an example, there are $9,057$ cases in the online YAGO where a child is born before her parent.

Differently from positive rules, literals play a vital role in discovering negative rules. In fact in many cases correct negative rules rely on temporal aspects in which something cannot happen before/after something else. Temporal information are usually expressed through dates, years, or other primitive types that are represented as literal values in KBs.

Last, discovering negative rules is faster than discovering positive rules. This is mostly due to the time we spend executing validation coverage queries. These queries are faster for negative rules since the validation set is just all the entities connected by the target predicate, whereas in the posi-

Table 3: Negative Rules Accuracy.

KB	*Avg. Run Time*	*# Pot. Errors*	*Precision*
DBPEDIA	19min, 40sec	499	**92.38%**
YAGO	10min, 40sec	2,237	**90.61%**
WIKIDATA	1h, 5min, 38sec	1,776	**73.99%**

tive case the validation set corresponds to counter examples described in Section 3.2, which are usually more complex to evaluate for standard SPARQL engines.

5.2 Comparative Evaluation

We compared the performance of our rules discovery method against AMIE [22], the state-of-the-art system in discovery Horn Rules from KBs.

AMIE is a rules discovery system designed to discover positive rules. It first loads the entire KB into memory, and then discovers positive rules for every predicate in the KB. The coverage of a rule is penalised with the partial closed world assumption, where the set of negative examples for a given pair (x, y) and a target predicate p is all those pairs where x is connected through p to an entity different from y. AMIE outputs all possible rules that exceeds a given threshold and ranks them according to their coverage function.

Given the in-memory implementation, AMIE cannot handle large KBs. We tried to run it on the KBs of Table 1, but the system goes quickly out of memory. Thus we downloaded and used the modified versions of Yago and DBPedia used in their experiments . These versions consist in the core facts of the KB, without literals and `rdf:type` facts. Table 4 summarises the characteristic of this dataset.

Table 4: AMIE Dataset characteristics.

KB	*Size*	*#Triples*	*#Predicates*	*#*`rdf:type`
DBPedia	551M	7M	10,342	22.2M
Yago	48M	948.3K	38	77.9M

Removing literals and `rdf:type` triples drastically reduce the size of the KB. Since our approach needs type information (both for the generation of G and V and for the discovery of entities inequality atoms), we run AMIE on its original dataset, while we run our algorithm on the same dataset plus `rdf:type` triples. The last column of Table 4 shows how many triples we added to the original AMIE dataset.

Positive Rules. We first compared RuDiK against AMIE on its natural setting: positive rules discovery. AMIE takes as input an entire KB, and discovers rules for every predicate in the KB. We adapted our system to simulate AMIE as follows: we first list all the predicates in the KB, and for each predicate that connects a *subject* to an *object* we computed the most common type for both subject and object. The most common type is the most popular `rdf:type` that is not super class of any other most popular type. We then run our approach sequentially on every predicate, setting $maxPathLen = 2$ (AMIE default setting).

AMIE outputs a huge amount of rules – 75 output rules in Yago, and 6090 in DBPedia. We followed their experiments setting and picked the first 30 best rules according to their score. We then picked the rules produced by our approach on the head predicate of the 30 best rules output of AMIE. RuDiK is more conservative and produces much less rules than AMIE. We noticed that for every predicate AMIE always discovers more than one rule, while there are several predicates where the output of our algorithm is empty since none of the plausible rules has enough support. As a consequence, for instance, RuDiK outputs just 11 rules for 8 target predicates on the entire Yago– for the remaining predicates RuDiK does not find any rule with enough support.

Figure 3 plots the total number of new unique predictions (x-axis) versus the aggregated precision (y-axis) on Yago.

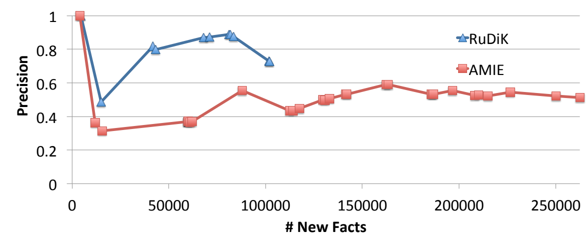

Figure 3: Predictions Accuracy on Yago

The n-th point from the left represents the total number of predictions and the total precision of these predictions, computed over the first n rules (sorted according to AMIE's score). AMIE produces many more predictions (262K vs 102K), but with a significant lower accuracy. This is due to the high number of rules in output of AMIE, but also to the way these rules are ranked. In fact if we limit the output of AMIE to the best 11 rules (same output of our approach), the final accuracy is still 29% below our approach, with just 10K more predictions. AMIE outputs many good rules that are preceded by meaningless ones in the ranking, and it is not clear how to set a proper k in order to get the best top k rules. RuDiK instead understands that in some cases meaningful rules do not exist, and it outputs something only when it has a strong confidence. This results in a lower number of predictions with a very high accuracy – precision is above 85% before the last rule, with more than 80K predictions.

Figure 4 shows the same evaluation on DBPedia. DBPedia has a richer set of relations, therefore also RuDiK is capable of producing 30 rules in output. Despite the same number of rules, once again our approach leads to a lower number of predictions (26K vs 41K) with a significant higher accuracy (85% vs 74%). The only point where AMIE outperforms our approach is when we consider the top 3 rules: the third rule discovered by AMIE is indeed a universally true rule that produces more than 11K correct predictions.

Negative Rules. As a second set of comparative experiments we used AMIE to discover negative rules. AMIE is not designed to work in this setting, and can discover rules only for predicates explicitly stated in the KB. Therefore we proceeded as follows: we sampled the top 5 most popular predicates in each KB and we created for each predicate a set of negative examples as explained in Section 3.2. For each negative example we added a new fact to the KB connecting the two entities with the *negation* of the predicate. The evaluation of negative rules is then carried out as explained before: we generate potential errors in the KB, and we manually evaluated the precision of such errors. Table 5 shows the results on the two KBs. RuDiK outperforms AMIE in both cases of almost 20%. This is because for the negation of a predicate, we use the actual predicate as counter examples. AMIE instead is not aware that the actual predi-

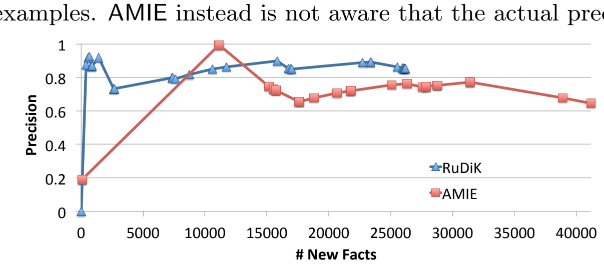

Figure 4: Predictions Accuracy on DBPedia

Table 5: Negative Rules vs AMIE.

	AMIE		RuDiK	
KB	*# Errors*	*Precision*	*# Errors*	*Precision*
DBPedia	457	38.85%	148	**57.76%**
Yago	633	48.81%	550	**68.73%**

cate provides counter examples for the negation of it, hence it is much less precise. In fact the output of AMIE consists in a high number of rules for each predicate (often more than 30), and in many cases AMIE produces same rules for both positive and negative scenarios.

Running Time. We report here the running time of AMIE compared to our approach. Note that numbers are different from [22], where AMIE was run on a 48 GB RAM server. AMIE could finish the computation only on Yago 2, while for other KBs it got stuck after some time. When this happened, we stopped the computation after we did not see any new rule in output for more than 2 hours.

Table 6 reports the running time on different KBs. The first five KBs are AMIE modified versions, while Yago 3* is complete Yago, including literals and `rdf:type`. The second column shows the total number of predicates for which AMIE produced at least one rule before getting stuck, while in brackets we report the total number of predicates. The third and fourth columns report the total running time of the two approaches. Despite being disk-based, RuDiK can successfully complete the task faster than AMIE in all cases, except for Yago 2. This is because of the very small size of the KB, which can easily fits in memory. However, when we deal with real KBs (Yago 3*), AMIE is not even capable of loading the KB due to out of memory errors. Eventually the last column reports the total time needed to compute `rdf:type` information for each predicate in the KB. This time is negligible w.r.t. the total running time.

Other Systems. We found other available systems to discover rules in KBs. [2] discovers new facts at instance level, hence less generic than our approach. On AMIE Yago 2 KB they discover 2K new facts with a precision lower than 70%. The best rule we discover on Yago 2 already produces more than 4K facts with a 100% precision. [11] implements AMIE algorithm with a focus on scalability. They do not introduce any novelty in the algorithmic part, but just a clever way of splitting the KB into multiple cluster nodes so that the computation can be run in parallel. The output is the same as AMIE. Eventually we did not compare with classic Inductive Logic Programming systems [15, 28], as these are already significantly outperformed by AMIE both in accuracy and running time.

5.3 Machine Learning Application

The main goal of this set of experiments is to prove the applicability of our approach in providing Machine Learning algorithms meaningful training examples. We chose DeepDive [32], a Machine Learning approach to incrementally construct KBs. DeepDive extracts entities and relations

Table 6: Run Time vs AMIE.

KB	*#Predicates*	*AMIE*	*RuDiK*	*Types*
Yago 2	20	30s	18m,15s	12s
Yago 2s	26 (38)	> 8h	47m,10s	11s
DBPedia 2.0	904 (10342)	> 10h	7h,12m	77s
DBPedia 3.8	237 (649)	> 15h	8h,10m	37s
Wikidata	118 (430)	> 25h	8h,2m	11s
Yago 3*	72	-	2h,35m	128s

from text articles via distant supervision. The key idea behind distant supervision is to use an external source of information (e.g., a KB) to provide training examples for a supervised algorithm. For example, the main showcase in DeepDive extracts mentions of married couples from text documents. In such a scenario DeepDive uses as a first step DBPedia in order to label some pairs of entities as *true* positive (those pairs of married couples that can be found in DBPedia). Unfortunately, a KB can only provide positive examples. Hence in DeepDive the burden of creating negative examples is left to the user through manual rules definition.

In this set of experiments we will use our negative rules on DBPedia to generate negative examples, and we will compare the output of DeepDive trained with different set of negative examples. We used DeepDive spouse showcase example . DeepDive already provides some negative rules to generate negative examples (e.g., if two people appear in a sentence connected by the words *brother* or *sister* then they are not married). We therefore compare the output of DeepDive using our generated negative examples and the ones generated with DeepDive rules.

Figure 5 shows DeepDive accuracy plot run on 1K input documents. The accuracy plot shows the ratio of correct positive predictions over positive and negative predications (y-axis), for each probability output value (x-axis). The dotted blue line represents the ideal situation, where the system finds high number of evidence positive predictions for higher probability output values – when the output probability is 0 there should not be positive predictions. The plot is computed over a test set, while the system is trained over a separated training set. The figure shows 4 lines other than the ideal ones. RuDiK is the output of DeepDive using our approach to generate negative examples. `OnlyPos` uses only positive examples from DBPedia, `Manual` uses positive examples from DBPedia and manually defined rules to generate negative examples, while `ManualSampl` uses only a sample of the manually generated negative examples in size equal to positive examples. The first observation is that `OnlyPos` and `Manual` do not provide valid training, as the former has only positive examples and labels everything as true, while the latter has many more negative examples than positive and labels everything as false. `ManualSampl` is the clear winner, while our approach suffers mostly the absence of data: over the input 1K articles, we could find only 20 positive and 15 negative examples from DBPedia. The lack of enough evidence in the training data also explains the missing points for RuDiK in the chart, where there are not predictions in the probability range 25-45%.

If we extend the input to 1M articles, things change drastically (Figure 6). All the three approaches except `OnlyPos` can successfully drive DeepDive in the training, with the

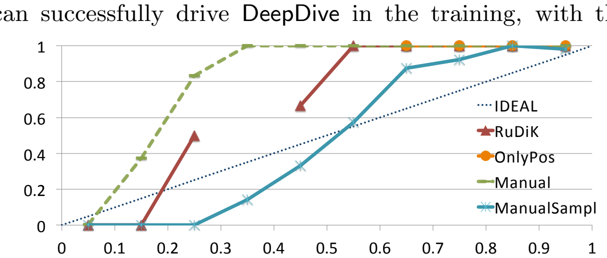

Figure 5: DeepDive Application 1K articles

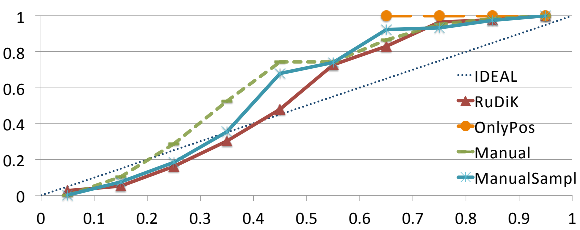

Figure 6: DeepDive Application 1M articles

examples provided with RuDiK leading to a slightly better result. This is because of the quality of the negative examples: our negative rules generate representative examples that can help DeepDive in understanding discriminatory features between positive and negative labels. The output of `ManualSampl` and RuDiK are very similar, meaning that we can use our approach to simulate user behaviour and provide negative examples at zero cost.

5.4 Internal Evaluation

In this last section we skecth the impact of individual components in RuDiK. Full results on the internal evaluation are not reported in this paper due to space reasons but can be examined on the technical report online at **TODO: add online link to technical report**.

Here we briefly outline some of the most relevant findings. *1)* We show the benefits of including literals in the mining especially for negative rules, where we double the number of errors discoverd with an almost 10% increase in precision. *2)* We show that 3 is the optimal value for the $maxPathLen$ parameter, since with shorter rules we loose several meaningful rules and with longer ones we do not gain anything in precision or errors detected. *3)* Other than $maxPathLen$ parameter, we empirically validate the choise for α and β for both positive and negative settings. *4)* We prove the validity of our LCWA negative examples generation, which clearly outperforms in accuracy random generation strategies. *5)* We quantify the benefits of A^* search algorithm and pruning, where on average the running time is halved. *6)* Eventually we show the benefits of our set cover problem formulation w.r.t. a ranking based one. We show that in almost every case good rules are not in the first best 10 ranked, and sometimes there exist good rules that are beyond the $100th$ best one.

6. RELATED WORK

Our work is inspired by Inductive Logic Programming, but uses techniques from dependencies discovery in relational databases, KBs construction, and mining graph patterns. We review a few of the most relevant works below.

A significant body of work has addressed the problem of discovering constraints over relational databases. Due to the presence of a pre-defined schema in relational databases, constraints are enforced based on the dependencies among the attributes by well-studied formalisms such as Functional Dependencies (FDs)([4], [3], [25], [37]), Conditional FDs ([20]) and Denial Constraints (DCs) [7,13]. It is interesting to note that the positive rules we discover using RuDiK carry a semantic resemblance to FDs while a close parallel can be drawn between our negative rules and DCs.

Despite being directly related to our output and expressing a richer language, FDs and DCs cannot be applied to RDF databases for three main reasons: *(i)* the schema-less nature of RDF data and the closed world assumption which no longer holds on RDF KBs; *(ii)* FDs and DCs rely on the assumption that data is either clean or has a negligible amount of errors; *(iii)* scalability issues on large RDF dataset: applying relational database techniques on RDF KBs would imply materializing all possible predicate combinations into relational tables.

A few FD mining approaches like [1, 26] that can work with erroneous datasets are still inapplicable to RDF data due to scalability problems. Fan et. al. [?] laid the theoretical foundations of Functional Dependencies on Graphs (GFDs). However, the language they propose covers only a portion of our negative rules to detect inconsistencies in graph databases and does not include smart literals comparisons, shown to be vital when detecting errors in KBs.

On the other hand, rule mining approaches specifically aimed towards RDF KBs such as AMIE [22] and OP algorithm [11] load the entire KB into memory prior to the graph traversal step connecting the predicates incrementally to form the full-fledged rule. In addition to this, neither of these two approaches can afford smart literal comparison. In contrast to them, RuDiK is disk-based and generates the graph on-demand and can discover rules on a small fraction of the KB examples that not only makes our approach scalable but also helps us accommodate smart literal comparisons from a language perspective. We have not tested our running time against [11] because we could not find an implementation of it, however it requires a powerful cluster of several machines to split the KB into multiple nodes. We showed in the experimental section how RuDiK outperforms AMIE both in final accuracy and running time.

Our example generation strategy in RuDiK leverages the popular Local Closed World Assumption (LCWA) to handle incompleteness in KBs. LCWA has been used in Google Knowledge Vault [17–19], AMIE [22] and helps us run with input example set of all sizes.

[2] is an instance-level approach that discovers new facts for specific entities contrary to which the rules generated by RuDiK are generic and can be instantiated with several instances thus being capable of generating several highly precise facts. [21] is a modern system to discover Conditional Denial Constraints (CDCs) from RDF Data. Differently from other systems, it includes literals in its language. CDCs can be directly mapped to our negative rules, however there is not a general correlation between CDCs and positive rules. Another major difference of our setting is the hardware: our disk-based approach is designed to handle large KBs with limited resources, while [21] works on a distributed environment with a total of 832 GB RAM memory.

To the best of our knowledge, RuDiK is the first approach that is generic enough to use the same algorithm to discover both positive and negative rules in RDF KBs.

ILP systems such as WARMR and ALEPH[1] are designed to work with a closed world assumption, and always require the definition of positive and negative examples. AMIE clearly outperfroms these two systems [22], showing evident scalability issues when dealing medium-size KBs. Moreover, one of the main limitations of classic ILP systems is the assumption of having high-quality, errors-free training examples. We showed how this assumption does not longer hold on in-

[1] `http://goo.gl/WcXSjy`

complete and erroneous KBs. Sherlock [31] is an ILP system that extracts first-order Horn Rules from Web text. While making RuDiK extensible to free text rather than relying on a well-defined KB can be seen as an interesting future work, the statistical significance estimates that is used by Sherlock need a threshold to discover meaningful rules. Several ILP systems inspired from Association Rule Mining [5] also use thresholds for support and confidence which are non-trivial to set (Section 5.2). We avoid the notion of a threshold RuDiK relies on a scoring function that simulates the set cover problem and outputs rules only when coverages over generation and validation sets are considered acceptable.

7. CONCLUSION

Extend rules with temporal and local information.

Combine positive and negative rules: if a positive and negative rule identifies same triples, then one of them must be wrong.

Introduce functions and smarter literals comparison: if two people are not born in the same century, then they cannot be married.

8. REFERENCES

[1] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning. *Proceedings of the VLDB Endowment*, 9(4):336–347, 2015.

[2] Z. Abedjan and F. Naumann. Amending rdf entities with new facts. In *European Semantic Web Conference*, pages 131–143. Springer, 2014.

[3] Z. Abedjan, P. Schulze, and F. Naumann. Dfd: Efficient functional dependency discovery. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 949–958. ACM, 2014.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. 22(2):207–216, 1993.

[6] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction for the web. In *IJCAI*, volume 7, pages 2670–2676, 2007.

[7] L. Bertossi. Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5):1–121, 2011.

[8] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. Dbpedia-a crystallization point for the web of data. *Web Semantics: science, services and agents on the world wide web*, 7(3):154–165, 2009.

[9] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. ACM, 2008.

[10] 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.

[11] Y. Chen, S. Goldberg, D. Z. Wang, and S. S. Johri. Ontological pathfinding. In *SIGMOD*, pages 835–846, 2016.

[12] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proceedings of the VLDB Endowment*, 6(13):1498–1509, 2013.

[13] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 458–469. IEEE, 2013.

[14] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[15] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and knowledge discovery*, 3(1):7–36, 1999.

[16] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: a report from the trenches. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1209–1220. ACM, 2013.

[17] 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 *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–610. ACM, 2014.

[18] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *Proceedings of the VLDB Endowment*, 7(10):881–892, 2014.

[19] X. L. Dong, E. Gabrilovich, K. Murphy, V. Dang, W. Horn, C. Lugaresi, S. Sun, and W. Zhang. Knowledge-based trust: Estimating the trustworthiness of web sources. *Proceedings of the VLDB Endowment*, 8(9):938–949, 2015.

[20] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):683–698, 2011.

[21] M. H. Farid, A. Roatis, I. F. Ilyas, H. Hoffmann, and X. Chu. CLAMS: bringing quality to data lakes. In *SIGMOD*, pages 2089–2092, 2016.

[22] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. Fast rule mining in ontological knowledge bases with amie+. *The VLDB Journal*, 24(6):707–730, 2015.

[23] P. S. GC, C. Sun, H. Zhang, F. Yang, N. Rampalli, S. Prasad, E. Arcaute, G. Krishnan, R. Deep, V. Raghavendra, et al. Why big data industrial systems need rules and what we can do about it. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 265–276. ACM, 2015.

[24] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[25] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[26] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1):129–149, 1995.

[27] B. Min, R. Grishman, L. Wan, C. Wang, and D. Gondek. Distant supervision for relation extraction with an incomplete knowledge base. In *HLT-NAACL*, pages 777–782, 2013.

[28] S. Muggleton. Inverse entailment and progol. *New generation computing*, 13(3-4):245–286, 1995.

[29] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

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

[31] 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*, pages 1088–1098. Association for Computational Linguistics, 2010.

[32] 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*, 8(11):1310–1321, 2015.

[33] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706. ACM, 2007.

[34] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from wikipedia and wordnet. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3):203–217, 2008.

[35] F. M. Suchanek, M. Sozio, and G. Weikum. Sofie: a self-organizing framework for information extraction. In *Proceedings of the 18th international conference on World wide web*, pages 631–640. ACM, 2009.

[36] D. Vrandečić and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[37] C. Wyss, C. Giannella, and E. Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 101–110. Springer, 2001.