Technical Report
VICKEY: Mining Conditional Keys on RDF datasets

Danai Symeonidou[1], Luis Galárraga[2], Nathalie Pernelle[3],
Fatiha Saïs[3], Fabian Suchanek[4]
[1]INRA, France [2]Aalborg University, Denmark
[3]LRI, France [4]Télécom ParisTech, France

Abstract

A conditional key is a key constraint that holds under certain circumstances – such as "the given name and the family name uniquely identify the members of the European parliament". In this paper, we show how such keys can be mined automatically on large knowledge bases (KBs). For this, we combine techniques from key mining with techniques from rule mining. We show that our method can scale to KBs of millions of facts. We also show that the conditional keys we mine can improve the quality of entity linking by up to 47 percentage points.

1 Introduction

Recent years have seen the rise of large knowledge bases (KBs), such as YAGO [24], Wikidata [27], and DBpedia [16] on the academic side, and the Google Knowledge Graph [6] or Microsoft's Satori graph on the commercial side. These KBs contain millions of entities (such as people, places, or organizations), and millions of facts about them. This knowledge is typically expressed in RDF [28], i.e., as triples of the form $\langle Obama, presidentOf, USA \rangle$. Some KBs provide an ontology, which describes the vocabulary (the classes and properties) for the RDF facts. The ontologies can also declare logical axioms on the data. We focus here on one particular type of axioms: key constraints. A key constraint specifies that no two distinct entities can share a certain set of properties (e.g., no two people share given name, family name, and birthdate). Key constraints are used for applications such as knowledge base fusion [7], knowledge base enrichment [19] and data linking [1, 21, 8].

When KBs are huge (with millions of triples and hundreds or thousands of properties) it is not feasible to specify the keys manually. Therefore, several approaches [25, 18, 2] have been developed to automatically discover the keys from RDF data. However, these works have also shown that for several datasets, there are no or only few keys that are valid in the entire dataset. This is the reason why we aim to mine *conditional keys* in this paper, i.e., keys that are valid in only a part of the data.

A conditional key is an axiom saying that under particular conditions, no two distinct entities can have the same values on a particular set of properties.

For example, we can say that at a German university, no two professors can advise the same doctoral student. The situation might be different at a French or American university – hence the key is "conditional" to German universities. This distinguishes conditional keys from classical keys, which hold under all circumstances. Thus, conditional keys can express constraints even in cases to which classical keys are blind. Hence, conditional keys are strictly more expressive than classical keys.

Conditional keys have many applications in the context of KBs. A first use is the debugging of the data. If, e.g., we find a German doctoral student with two advisors, we can flag this case for human review or for additional sourcing. In this way, we can spot potential errors and clean up the data – all while focusing our effort on cases that are likely to be problematic. A second application is the detection of duplicates. If we find two professors in Germany with a similar name who have the same student, then this can be a signal that the two professors are just one person. This can then help removing duplicates, and consolidating the data. Again, we expect conditional keys to find more of such matching opportunities than classical keys. A third use is the linking of data. Given two KBs with different identifiers for the same person, we can use the fact that two professors advise the same student as a signal for matching them. In this way, we can link the KBs, and thus allow queries that span two sources. Finally, conditional keys carry value in themselves. For example, when we compare national policies, it is interesting to know that France allows several advisors, while Germany does not.

Mining conditional keys automatically from the data is a challenging endeavor, for several reasons. First, KBs contain only binary relations, which means that keys can potentially span a dozen relationships. This distinguishes our scenario from key mining in relational databases [22, 13], which are relation-centric. Second, KBs are usually incomplete [20]. If a student at a German university has only one advisor in the KB, she could still have several in real life. Thus, any approach that automatically mines conditional keys risks being misled. Finally, the challenge is scale: Today's KBs contain millions of statements. This means that there are billions of possible conditions and property combinations that could define a conditional key. In the example, professors could be distinguished by their doctoral students, but also by their given and family name or by their discipline and birthdate. These key constraints could hold only for German professors, only for Danish ones, only for professors at a certain university in Mexico, or only for professors born in a certain city in Iowa. This huge search space is one of the main reasons why there is today no approach that could mine conditional keys on KBs.

Our idea is to combine techniques from key mining [25] with techniques from rule mining [12]. Rule mining is concerned with finding dependencies in the data, such as "If a person is born in a city that is located in a country, then usually the person has the citizenship of that country". We show that this technique can be exploited to mine the conditions under which a key could hold. More precisely, our contributions are as follows:

- We develop an algorithm that can mine conditional keys efficiently.
- We show that our method scales gracefully to KBs of millions of facts.
- We show that the use of our conditional keys improves the F1 measure of

KB linking by up to 47 percentage points over the use of classical keys.

The rest of this paper is structured as follows. We discuss related work in Section 2, and introduce preliminaries in Section 3. In Section 4 we present our approach. Section 5 showcases our experiments, before Section 6 concludes.

2 Related Work

Several approaches have been developed to discover integrity constraints (ICs) in relational databases or semantic (RDF) data. Some are focused on functional dependencies, others are more specific and aim at finding keys. In the following, we summarize the most relevant work in this area, both for relational data and for RDF data.

Relational Data. A functional dependency (FD) is a relationship between two sets of attributes, such that the values of the first set identify uniquely the values of the second. For instance, given a relation with zipcodes and cities, the city is unambiguously determined by the zipcode. A key is a particular type of FD, where the second set of attributes is a unique identifier for the record in the database. Hence, approaches that can mine FDs [14, 31, 15, 3, 30] can in principle also mine keys.

Some approaches have focused on more expressive ICs such as conditional functional dependencies (CFDs). A CFD expresses a functional dependency that holds on a subset of tuples satisfying a given pattern or condition [9, 4]. The problem of CFD discovery was first addressed in [4]. This approach discovers minimal CFDs using a strategy inspired by the level-wise schema driven approach TANE [14], designed to discover functional dependencies. Such an approach can detect that when two customers are in the UK (country code 44), the zipcode uniquely determines the city.

Other works focus on Denial Constraints (DCs). A DC is a universally quantified conjunctive expression in first-order logic that states that all its predicates cannot be true at the same time. An example is "among members of the same university the salary of a post-doc cannot be greater than the salary of a professor", $\neg(Professor(x) \wedge Postdoc(y) \wedge univ(x,z) \wedge univ(y,z) \wedge salary(x) < salary(y))$. The work of [5] proposes a general algorithm to discover DCs from a relational database. The authors have also shown that some of the discovered DCs express composite key constraints. In this approach the discovered DCs can exploit constants and numerical built-in operators (as in $salary(x) < salary(y)$).

These approaches cannot be applied directly to RDF data, for several reasons. First, key discovery in databases is geared towards relations that contain one single value for each subject. RDF relations, in contrast, can contain one or several values for the same subject. Second, relational databases operate under the Closed World Assumption (CWA), i.e., they assume that the database is complete. RDF, in contrast, operates under the Open World Assumption (OWA), i.e., unknown assertions are not necessarily false. In practice, RDF KBs are highly incomplete. Finally, RDF data usually comes with an ontology, which specifies additional semantic constraints that have to be taken into account. For all of these reasons, keys have been defined differently for databases and for RDF data [29]: A database key fires when two entities share *all* values for each property. An RDF key, in contrast, fires as soon as two entities share *at least one* value for each property.

RDF Data. Several approaches have been proposed to mine keys on RDF data [2, 23, 18, 26, 25]. Some of these approaches discover keys only under the Closed World Assumption, i.e., if all the property values are known for each instance [2, 23]. In such approaches, sets of property values can be efficiently compared since only equal sets of property values will lead to an identity link. Other approaches [18, 25] aim at discovering keys even if not all property values are known. To avoid scanning the entire dataset, these approaches discover first the maximal non-keys and then derive the keys from this set. However, none of these approaches [2, 23, 18, 26, 25] can mine *conditional* keys.

One way to mine conditional keys would be to resort to logical rule mining. For example, the conditional key that the firstname (fn) and the lastname (ln) uniquely identify people who live in Paris (li) can be expressed as the rule:

$$li(x, Paris) \wedge li(y, Paris) \wedge fn(x, f) \wedge fn(y, f) \wedge\ ln(x, l) \wedge ln(y, l) \Rightarrow equals(x, y)$$

Thus, conditional keys could be found by mining rules of this type. The AMIE system [12, 11] can learn logical rules on RDF KBs that contain millions of triples. The system scales to rules with up to four atoms. However, even relatively simple conditional keys can easily contain five or more atoms, as the example shows. This leads to an exponential increase of the search space that current rule mining approaches cannot handle. Besides, a generic rule mining approach cannot benefit from prunings and optimizations that are specific to the problem of key mining. We show indeed in our experiments that AMIE cannot mine conditional keys efficiently.

3 Preliminaries

3.1 Datasets and Keys

Datasets. We consider in this paper RDF[1] knowledge bases. RDF is a W3C recommendation [28] that has become the de facto data model to represent semantic data. This data model is based on a set $\mathcal{I}$ of instances (such as *France*, the research institute *INRA*, or a person identified as *p1*), a set $\mathcal{L}$ of literals (such as strings and numbers), a set $\mathcal{P}$ of properties (such as *firstName* or *nationality*), and a set $\mathcal{C}$ of class names (such as *country* or *researcher*). A fact is a triple of a subject $s \in \mathcal{C} \cup \mathcal{I}$, a predicate $p \in \mathcal{P}$, and an object $o \in \mathcal{C} \cup \mathcal{I} \cup \mathcal{L}$, which we write as $p(s, o)$. Examples are *gender(p1, male)* or *firstName(p1, "Claude")*. Every instance is typically associated to one or more classes by the *type* property (as in *type(p1, researcher)*), and these classes can be arranged in a hierarchy by the *subclassOf* property. We assume that every instance of a class is also explicitly stored as an instance of all superclasses. A set of such facts is called a knowledge base (KB).

Given a KB $\mathcal{K}$, a *dataset* for a class c of $\mathcal{K}$ is the set of all facts that have an instance of c as subject or object. Facts $p(x, y)$ that have the instance as the object are rewritten as $p^{-1}(y, x)$, where p^{-1} is a name for the inverse relationship of p, so that the instances of c appear always as subjects.

Definition 1. ***(Dataset)*** *The dataset of a class c in a knowledge base $\mathcal{K}$ is the set $\{p(x, y) \in \mathcal{K} : type(x, c) \in \mathcal{K}\} \cup \{p^{-1}(x, y) : type(x, c) \in \mathcal{K} \wedge p(y, x) \in \mathcal{K}\}$.*

[1] Resource Description Framework

	FN	LN	Gender	Lab	Nat
p1	Claude	Dupont	Female	Paris-Sud	France
p2	Claude	Dupont	Male	Paris-Sud	Belgium
p3	Juan	Rodríguez	Male	INRA	Spain
p4	Juan	Salvez	Male	INRA	Spain
p5	Anna	Papadopoulou	Female	INRA	Greece
p6	Pavlos	Georgiou	Male	Paris-Sud	Greece
p7	Marie	Legendre	Female	INRA	France

Table 1: Example dataset

Table 1 shows an example dataset about researchers $p1, ..., p7$, who each have the properties *firstName* (FN), *lastName* (LN), *gender*, *lab*, and *nationality* (Nat). When a dataset $\mathcal{D}$ is given, we write $p(x, y)$ to mean $p(x, y) \in \mathcal{D}$.

Keys. A key for a dataset is a set of properties that uniquely identifies every subject.

Definition 2. ***(Key)*** *A (classical) key for a dataset $\mathcal{D}$ is a set of properties $p_1, ..., p_n$ of $\mathcal{D}$, such that for all subjects x, y of $\mathcal{D}$, $(\bigwedge_{i=1...n} \exists\ u_i :\ \ p_i(x, u_i) \wedge p_i(y, u_i)) \Rightarrow x = y$.*

This definition of keys is used in the ontology language OWL2 [29]. In our example, the property set {*lastName*, *gender*} is a key. Every superset of a key is a key as well. This is easy to see: Once {*lastName*, *gender*} uniquely identifies all researchers, the addition of the nationality to the key will only further distinguish the subjects. Therefore we are mainly interested in minimal keys, i.e., keys where the removal of any of the properties produces a set that is no longer a key, i.e, a non-key.

Definition 3. ***(Maximal non-key)*** *A maximal non-key for a dataset $\mathcal{D}$ is a set of properties P of $\mathcal{D}$ that is not a key, so that the addition of any property makes P a key.*

In our example, {*firstName, lastName, laboratory*} is a maximal non-key, because adding any other property makes the set a key.

SAKey approach for key discovery. One challenge in the context of KBs is to discover keys automatically in the data. This is a difficult endeavor, because KBs can be very large, and all different combinations of properties have to be checked for their quality of being a key. To check a combination of properties, a naive algorithm has to compare all subjects of the dataset to all other subjects – which is prohibitively expensive. SAKey[2][25] is an algorithm for key discovery that avoids this complexity. Instead of directly finding keys, SAKey first finds maximal non-keys. This is more efficient, because to verify that a set of properties is a non-key, it suffices to find two subjects that share the properties. SAKey starts with property combinations that contain only a single property, and incrementally adds more and more properties until it arrives at maximal non-keys. When it has found all non-keys, all other property combinations must be keys – which is what SAKey outputs.

[2]Scalable Almost Key discovery

3.2 Conditional Keys

In our example in Table 1, there exist two researchers with the last name "Dupont", therefore the property *lastName* is not a key in this dataset. The combination {*firstName, lastName*} is also not a key, since there are two researchers with the same first and last names. However, when we restrict our set of researchers to those working at INRA, the property *lastName* identifies researchers uniquely. In contrast, this is not true for the researchers in Paris-Sud. Thus, {*lastName, lab*} is not a key in general. We say that *lastName* is a *conditional key* for people working at INRA. More formally:

Definition 4. ***(Condition)*** *For a given dataset $\mathcal{D}$, a condition is a pair of a property p of $\mathcal{D}$ and an object o of $\mathcal{D}$, written $p = o$. A condition cd with property p and object o holds for a subject x, written $cd(x)$, if $p(x, o)$.*

In our example, $lab = INRA$ is a condition, and it holds for $p4$, because $lab(p4, INRA)$.

Definition 5. ***(Conditional key)*** *A conditional key for a dataset $\mathcal{D}$ is a non-empty set of conditions $\{cd_1, ..., cd_n\}$ and a non-empty set of properties $\{p_1, ..., p_m\}$ of $\mathcal{D}$ (disjoint from the properties in the conditions), such that for all subjects x, y:*

$$\bigwedge_{i=1..n} (cd_i(x) \wedge cd_i(y)) \quad \wedge \quad \bigwedge_{i=1..m} (\exists\ u_i : \quad p_i(x, u_i) \wedge p_i(y, u_i)) \Rightarrow x = y$$

Definition 6. ***(Minimal conditional key)*** *A conditional key with conditions CD and properties P is minimal, if the removal of a condition, the removal of a property, or the addition of a property p to P at the expense of removing a condition with property p from CD all result in something that is not a conditional key.*

In our example, *lastName* is a conditional key for Spanish researchers at INRA, but this conditional key is not minimal, because there exists a simpler version of the key with fewer conditions (i.e., being only Spanish). In the same vein, {*lastName*} with the condition *gender=male* is not a minimal conditional key, because {*lastName, gender*} is a key.

Definition 7. (Support of a conditional key). *The support of a conditional key with properties $\{p_1, ..., p_n\}$ and conditions $\{cd_1, ..., cd_m\}$ for a dataset $\mathcal{D}$ is the number of subjects x such that $\bigwedge_{i=1..n} \exists u_i : p_i(x, u_i)$ and $\bigwedge_{i=1..m} cd_i(x)$.*

The support is an absolute number. A proportional version of the support, which we call the *coverage*, measures the ratio of subjects in the dataset identified by the conditional key. In our example, the support of the key {*lastName*} under the condition *lab=INRA* is 4, and the coverage is $\frac{4}{7} = 0.57$ since there are 7 subjects in the dataset.

Relation to OWL. Conditional keys can also be defined in the ontology language OWL2 [29]. This is because OWL2 allows defining keys not just on atomic classes (such as *researcher*), but also on more complex class expressions. We can, e.g., define the class "Researchers who work at INRA" as $c = Researcher \sqcap \exists lab.\{INRA\}$. Then {*lastName*} is a key on the dataset of c according to Definition 2.

4 Mining Conditional Keys

We now present our approach to automatically discover conditional keys in RDF datasets. The discovery of simple keys alone already requires checking a large number of property combinations (of which there are $2^{|\mathcal{P}|}$ in total, where $\mathcal{P}$ is the set of properties). Discovering conditional keys is even more complex, since the search space is in the order of $O(|\mathcal{V}|^{|\mathcal{P}|})$, where $\mathcal{V}$ is the set of objects in the dataset. Our algorithm can discover conditional keys efficiently in spite of this large search space. Our method takes as input a dataset $\mathcal{D}$ and a threshold θ for the minimal support of the discovered keys. We proceed in three phases:

Discovery of non-keys. Instead of exploring the whole set of combinations of properties, we focus our search on those combinations that are not keys.

Generation of Conditional Key Graphs. The non-keys from the previous phase are used to generate candidate keys, which we store in *conditional key graphs*.

Mining of Conditional Key Graphs. The conditional key graphs are then mined for minimal conditional keys.

We will now explain these phases in detail.

4.1 Discovery of non-keys

The naive method to mine conditional keys explores all possible combinations of properties and conditions in the input KB and verifies whether they fulfill Definition 5. Such an approach is infeasible on large datasets. Our main idea (the key insight, so to speak) is the following:

Observation 1. ***(Conditional Keys and Non-Keys)*** *Given a minimal conditional key for a dataset $\mathcal{D}$ with properties P and conditions $\{p_1 = o_1, ..., p_n = o_n\}$, the set of properties $P \cup \{p_1, ..., p_n\}$ must be a non-key for $\mathcal{D}$.*

This follows from Definition 6. In our example from Table 1, {*firstName*} is a minimal conditional key with condition *gender=Female*, and {*gender, firstName*} is a non-key.

Thus, if we want to mine the complete set of minimal conditional keys, it suffices to consider only the property combinations given by non-keys. Since maximal non-keys are super-sets of all other non-keys (Definition 3), it is sufficient to explore only property combinations given by maximal non-keys. The maximal non-keys in the input dataset can be mined efficiently with the SAKey algorithm [25] (Section 3.1). Thus, we concentrate in the following on mining the conditional keys from these maximal non-keys.

As a running example, consider again the dataset in Table 1. It contains two maximal non-keys: {*firstName*, *lastName*, *lab*} and {*firstName*, *gender*, *lab*, *nationality*}.

4.2 Generation of Conditional Key Graphs

Our method for discovering conditional keys from non-keys relies on a data structure that we call *conditional key graph*. Such a graph is a tuple $\langle P^k, P^c, cond, G \rangle$

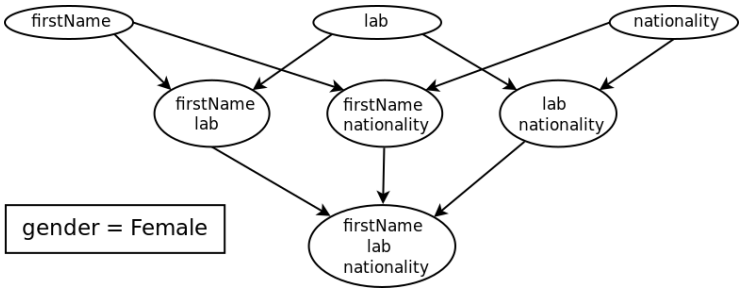

Figure 1: Example of a conditional key graph with $P^k = \{firstName,\ lab,\ nationality\}$, $P^c = \{gender\}$, $cond = \{gender = Female\}$.

with the following components:

- P^k and P^c are disjoint sets of properties, called *key properties* and *condition properties*, respectively.
- *cond* is a set of conditions (Definition 4) on P^c.
- G is a directed graph. Each node v is associated to a set $v.p \subseteq P^k$ and to a boolean flag $v.explore$. There is a directed edge from u to v if $u.p \subset v.p$ and $|u.p| = |v.p| - 1$.

As an example, Figure 1 shows a conditional key graph with $P^k = \{firstName, lab, nationality\}$, $P^c = \{gender\}$, and $cond = \{gender{=}Female\}$.

We construct the initial conditional key graphs with Algorithm 1. This algorithm takes as input the dataset, the support threshold, and the non-keys discovered in Section 4.1. We first construct all possible conditions $p = a$ that combine a property p from the non-keys with an instance or literal a from the dataset (Lines 2-3). Conditions with support less than θ are not considered (Line 4). We then look at all non-keys N in which p appears (Line 5). The conditional key graph for the condition $p = a$ will contain as nodes all subsets of $N \setminus \{p\}$ (Lines 6-8) except the empty set (Line 9).

As an example, let us consider again the dataset of Table 1 and its two maximal non-keys {*firstName*, *lastName*, *lab*} and {*firstName*, *gender*, *lab*, *nationality*}. Figure 1 depicts the conditional key graph associated to the condition *gender* = *Female* constructed by Algorithm 1.

Observation 2. ***(Graph Construction)*** *If Algorithm 1 is given a dataset $\mathcal{D}$, a complete set of maximal non-keys $\mathcal{N}$ for $\mathcal{D}$ and a support threshold θ, then for each conditional key of $\mathcal{D}$ with a single condition and with at least support θ, there is a graph in the output that contains the key condition and a node with the key properties.*

Observation 2 follows from the fact that Algorithm 1 (a) iterates over all conditions $p = o$ with a least support θ and (b) considers *all* the possible subsets of properties $2^{N \setminus \{p\}}$ with $N \in \mathcal{N}$ (except for $\emptyset$). From Observation 1 we recall that for any conditional key with properties P and condition $p = o$, the set of properties $P \cup \{p\}$ must be a non-key. Thus, from the completeness of our set of maximal non-keys, it follows that our graph contains in its nodes all possible keys with support higher than θ for a given condition $p = o$.

Algorithm 1: ConstructGraphs

Input: dataset $\mathcal{D}$, min. support θ, set of non-keys $\mathcal{N}$
Output: set of conditional key graphs $\mathcal{G}$

1 $\mathcal{G} \leftarrow \emptyset$
2 **for** $p \in \bigcup_{N \in \mathcal{N}} N$ **do**
3 **for** $a \in \mathcal{I} \cup \mathcal{L}$ **do**
4 **if** *number of* x *with* $p(x, a)$ *is at least* θ **then**
5 $V \leftarrow \emptyset$
6 **for** $N \in \mathcal{N}$ *where* $p \in N$ **do**
7 $V \leftarrow V \cup 2^{N \setminus \{p\}}$
8 $V \leftarrow V \setminus \emptyset$
9 $E \leftarrow \{u, v \in V : u.p \subset v.p \wedge |u.p| = |v.p| - 1\}$
10 $P^k = \bigcup_{v \in V} v.p$
11 $P^c = \{p\}$
12 $cond = \{p = a\}$
13 $\mathcal{G} \leftarrow \mathcal{G} \cup \{\langle P^k, P^c, cond, (V, E)\rangle\}$
14 **return** $\mathcal{G}$

4.3 Mining of Conditional Key Graphs

Mining conditional keys. Algorithm 2 takes as input a dataset, a support threshold, and the set of conditional key graphs constructed in the previous phase. All these conditional key graphs have conditions of size 1 (see Algorithm 1). The algorithm proceeds in batches, looking first at the graphs with condition size 1, then size 2, etc. (Lines 2-3). For each batch, it mines the conditional keys (Line 7). The graphs in one batch are then post-processed (Line 8) to give rise to new graphs with conditions of larger size. The algorithm iterates until all sizes are processed (Line 5).

Algorithm 2: ConditionalKeyDiscovery

Input: dataset $\mathcal{D}$, minimum support θ, set of conditional key graphs $\mathcal{G}$
Output: set of minimal conditional keys CKs

1 $CKs \leftarrow \emptyset$
2 **for** $size = 1$ *to* ∞ **do**
3 $\mathcal{G}' \leftarrow \{g \in \mathcal{G} : |g.cond| = size\}$
4 **if** $\mathcal{G}' = \emptyset$ **then**
5 **return** CKs
6 **for** $\langle P^k, P^c, cond, G\rangle \in \mathcal{G}'$ **do**
7 $CKs \leftarrow CKs \cup MineGraph(\mathcal{D}, \theta, \langle P^k, P^c, cond, G\rangle, CKs)$
8 $\mathcal{G} \leftarrow newConditions(size, \mathcal{G}, \theta, \mathcal{D})$

Mining a conditional key graph. Let us now discuss how one conditional key graph can be mined for keys (Line 7 in Algorithm 2). This task is done by Algorithm 3. This algorithm takes as input a dataset, the support threshold, a conditional key graph, and the set of conditional keys found so far. The

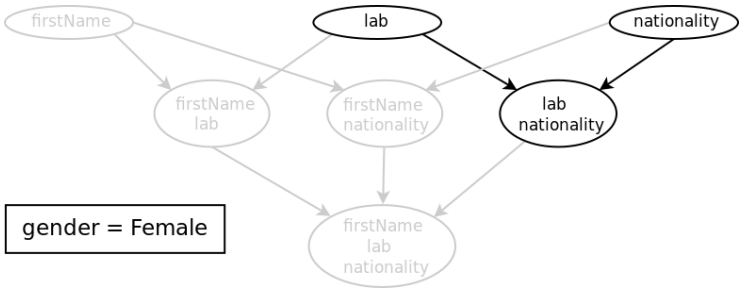

Figure 2: Keys of size 1 explored for the condition $gender = Female$.

algorithm proceeds in levels, looking first at the nodes that contain one property, then two properties, etc. For each level, we consider every node *cand*. If the node is still marked for exploration (Line 3), we construct a candidate conditional key, with the input conditions as condition part, and the properties in $cand.p$ as the key part (Line 4). We then verify if the candidate key (a) meets the definition of a conditional key and (b) is minimal with respect to the other keys that have already been mined (Lines 5-6). If that is the case, the conditional key is added to the ouput (Line 7). If the key is a minimal key, then any extension of the key with more properties in the key part must be non-minimal and can safely be abandoned. Likewise, if the support of the candidate key is below the given threshold, so are its refinements. In both cases, we can prune the node and all descendants (Lines 8-11).

As an example, let us consider again the data from Table 1, with the condition *gender=Female* and the maximal non-key {*firstName*, *gender*, *lab*, *nationality*}. The corresponding conditional key graph after scanning the first level is shown in Figure 2. Nodes with the *explore* flag set to false are greyed out. At the end of this step, only the property *firstName* is discovered as a key, since first names are unique among female researchers. It follows that nodes containing this property in the next levels of the graph define non-minimal keys. They are therefore discarded for further exploration (the *explore* flag is set to false). The search for conditional keys is then applied to the nodes on levels 2 and 3, for which the *explore* flag is still true.

Observation 3. ***(Graph Mining)*** *Given a conditional key graph $\langle P^k, P^c, cond, G\rangle$ for a dataset $\mathcal{D}$ and a threshold θ, Algorithm 3 will ensure that the result set CKs contains all minimal conditional keys for the condition set cond whose key properties are given by one of the nodes in G, and whose support is at least θ.*

This observation holds because Algorithm 3 traverses all nodes in the conditional key graph, and checks each of them for being a conditional key. It excludes only (a) those nodes whose ancestors already had a support smaller than θ, in which case the node itself must also have a support smaller than θ, and (b) the nodes that lead to non-minimal keys.

Merging conditions. Let us now look at the process of generating more complex conditions (Line 8 in Algorithm 2). This work is done by Algorithm 4. It takes as input a set of conditional key graphs, a support threshold, a dataset, and a size parameter. It looks at all conditional key graphs that have a condition set of the given size (Lines 2-3). For each of them, it constructs a clone (Line

Algorithm 3: MineGraph

Input: dataset $\mathcal{D}$, minimum support θ,
conditional key graph $\langle P^k, P^c, cond, G = (V, E)\rangle$,
set of conditional keys found so far CKs
Output: modified CKs

```
for level = 1 to max_{v∈V} |v.p| do
    for cand ∈ V where |cand.p| = level do
        if cand.explore then
            ck ← ⟨cond, cand.p⟩
            isKey ← ck is a minimal key w.r.t CKs
            if isKey ∧ support(ck, D) ≥ θ then
                CKs = CKs ∪ {ck}
            if isKey ∨ support(ck, D) < θ then
                cand.explore ← false
                for child ∈ descendants(cand, G) do
                    child.explore ← false
return CKs
```

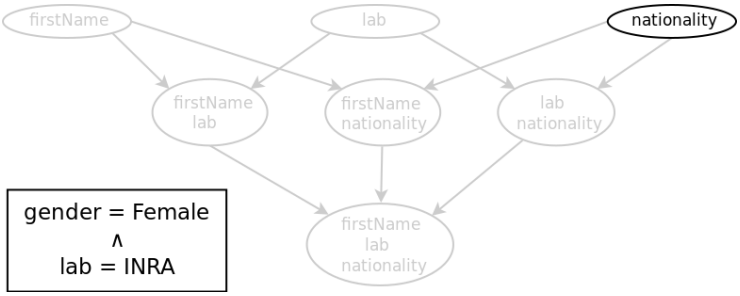

Figure 3: Example of a merged graph with condition $\{gender = Female, lab = INRA\}$.

4). It then adds one more condition to the condition set of the clone. This new condition consists of a property and a constant (Definition 4). The property is taken from a node of size 1 having its *explore* flag still set to true (Lines 5-6). The constant is taken from the constants that appear with that property in the conditional key graphs of size 1 (Line 10). If the new combined condition has a support that is large enough (Line 11), the conditional key graph of the singleton condition is merged with the clone and added to the output set (Line 12).

The *merge* operation between two conditional key graphs $\langle P_1^k, P_1^c, cond_1, (V_1, E_1)\rangle$ and $\langle P_2^k, P_2^c, cond_2, (V_2, E_2)\rangle$ with $P_1^c \cap P_2^c = \emptyset$, produces a new conditional graph $\langle P^k, P^c, cond, (V, E)\rangle$ with:

- $P^k = P_1^k \cap P_2^k$ and $P^c = P_1^c \cup P_2^c$.
- $cond = cond_1 \cup cond_2$
- $V = \{\langle v.p, v.explore\rangle : \exists v_1 \in V_1, v_2 \in V_2 : v_1.p = v_2.p = v.p \wedge v.explore = (v_1.explore \wedge v_2.explore)\}$
- $E = \{u, v \in V : u.p \subset v.p \wedge |u.p| = |v.p| - 1\}$

Algorithm 4: newConditions

Input: size of condition set *size*
set of conditional key graphs $\mathcal{G}$
support threshold θ, dataset $\mathcal{D}$
Output: modified set of conditional key graphs $\mathcal{G}$

```
1  G_1 ← {g ∈ G : |g.cond| = 1}
2  G_size ← {g ∈ G : |g.cond| = size}
3  for g ∈ G_size do
4      g ← clone(g)
5      for v ∈ g.V where |g.V.p| = 1 do
6          if v.explore then
7              v.explore ← false
8              for v' ∈ g.descendants(v) do
9                  v'.explore ← false
10             for g_1 ∈ G_1 where g_1.P^c = v.p do
11                 if support(g.cond ∧ g_1.cond, D) ≥ θ then
12                     G ← G ∪ merge(g, g_1)
13 return G
```

As an example, Figure 3 shows the conditional graph with the set of conditions $\{gender = Female, lab = INRA\}$ produced by Algorithm 4 from the conditional graphs with conditions $gender = Female$ and $lab = INRA$. This graph is a clone of the graph with the condition $gender = Female$. A node is marked to be explored only if it was marked to be explored in both of the original graphs.

Observation 4. ***(New Conditions)*** *Given a dataset $\mathcal{D}$, a set of conditional key graphs $\mathcal{G}$, a size parameter size, and a threshold θ, Algorithm 4 produces all conditional key graphs that contain condition sets of size $size + 1$. Each of those graphs contains all the conditional keys for the given condition.*

We can prove Observation 4 by induction. For $size = 0$, Observation 2 guarantees that Algorithm 4 starts with all conditional key graphs for conditions of size 1. For $size > 0$, we need to show that (a) Algorithm 4 generates all conditional key graphs of size $size + 1$ and (b) each of these graphs contains all minimal conditional keys for their condition.

We start by showing (b), that is, the merge operation between two conditional key graphs $G_1 = \langle P_1^k, P_1^c, cond_1, (V_1, E_1)\rangle$ and $G_2 = \langle P_2^k, P_2^c, cond_2, (V_2, E_2)\rangle$ does not skip any minimal conditional key for the new condition. There are only two ways a node can be excluded from exploration in the merge operation: (1) the node is explicitly marked for non-exploration and (2) the node does not occur in one of the conditional key graphs. Case (1) occurs when the corresponding nodes are below the support threshold θ or they define non-minimal keys. In Case (2), the claim follows from the fact that if a node v is not contained in one of the graphs (e.g., $v \notin V_1$), then $v.p \cup P_1^c$ must be a key, i.e., it is not contained in any maximal non-key. This rationale applies analogously if $v \notin V_2$.

To show (a) we need to prove that our conditions are complete and correct. To show completeness we observe that Algorithm 4 builds conditions with $|cond| = size + 1$ based on the complete set of conditions with $|cond| = size$ and $|cond| = 1$. From the monotonicity of support, it follows that all conditions with $|cond| = size + 1$ with support greater than θ can be computed from these sets. To show correctness we note that for each graph with condition $cond = \{p_1 = o_1, \ldots, p_{size} = o_{size}\}$ and key properties P^k, Algorithm 4 will merge the graph with all conditions of the form $p_{size} = o_{size}$ where $p_{size} \in P^k$ (conditions of size 1, Line 5). This will produce graphs with conditions of the form $\{p_1 = o_1, \ldots, p_{size-1} = o_{size-1}, p_{size} = o_{size}\}$ with key part $P^k \setminus \{p_{size}\}$ with support greater than θ. (a) follows from Observation 1, since we have just transferred a property from the key part to the condition part.

Theorem 1. ***(Conditional Key Discovery)*** *Given a dataset $\mathcal{D}$, a set of conditional key graphs $\mathcal{G}$, and a threshold θ, Algorithm 2 produces all conditional keys whose properties are a subset of the properties of any node in any graph in $\mathcal{G}$, whose conditions are built from conditions or properties in $\mathcal{G}$, and whose support is at least θ.*

This theorem follows from the fact that Algorithm 2 calls Algorithm 3 for all sizes between 1 and the maximal number of property combinations. Observation 3 makes sure that all possible graphs are generated. Observation 4 ensures that all possible combinations of conditions are treated.

Corollary 1. ***(Conditional Key Mining)*** *Our method for conditional key mining is complete and correct.*

The correctness follows from the fact that Algorithm 3 adds a new key if and only if it is a key (Line 5). The completeness follows from Observation 1 and Theorem 1.

4.4 Implementation

Our method, VICKEY, is implemented in Java 7. The conditional key graphs have large condition sets and large associated graphs. Therefore, we do not

Class	Triples	Instances	#Properties	#NKs	VICKEY	AMIE	#CKs
Actor	74.0k	5.9k	133	244	8.83m	> 1d	401
Album	831.3k	89.0k	106	93	1.83h	7.82h	295
Book	262.6k	30.1k	117	132	14.44h	> 1d	411
City	921.5k	19.3k	283	1329	> 2d	> 2d	> 93
Film	839.3k	89.2k	172	232	1.80h	4.86h	184
Mountain	133.3k	16.5k	140	78	3.51m	> 1d	257
Museum	14.4k	2.1k	123	29	1.46s	11.77s	51
Organisation	4.01M	191.3k	776	4624	37.53h	> 2d	25
Scientist	559.7k	19.7k	186	443	33.53m	> 1d	575
University	191.4k	10.9k	238	409	12.45h	> 1d	805

Table 2: Performance of VICKEY vs AMIE on different DBpedia classes

store the graphs in memory, but rather generate them on the fly when they are accessed. Furthermore, we parallelized the algorithm as follows: the set of input non-keys is split into batches containing up to 50 (potentially non-distinct) properties. The batches are then scheduled to threads in the system, each one running Algorithms 1 and 2. This may lead to mining the same non-key multiple times, and therefore we perform a de-duplication before reporting the final results.

5 Experiments

We conduct two rounds of experiments to evaluate VICKEY. In the first round, we evaluate the performance of VICKEY and compare it with a conditional key mining approach based on AMIE [11, 12]. In the second round, we show how conditional keys can improve the quality of the entity linking task between two KBs.

For the first round of experiments, we use the version of DBpedia released in October 2015[3]. We use the instance information and the mapping-based properties. In the second round of experiments, we use the conditional keys found in the first round to link instances between DBpedia and YAGO3[4]. Since the notion of a key is associated to a class, we test VICKEY on a set of 10 classes, covering different domains such as people, organisations, and locations. We construct one dataset per class (Definition 1). Our experiments are run on a server with an AMD Opteron 6376 Processor (2.40GHz), 8 cores, and 128GB of RAM.

5.1 VICKEY Runtime

Setting. We evaluate the performance of VICKEY by comparing it with a generic rule mining approach, AMIE [11, 12]. Given an input KB and a threshold on support, AMIE mines Horn rules of the form $B_1 \wedge \cdots \wedge B_n \Rightarrow H$, like:

$$hasChild(x, y) \wedge isCitizenOf(x, z) \Rightarrow isCitizenOf(y, z)$$

We adapted AMIE to mine rules of the form:

$$P_c \wedge P_k \Rightarrow x = y$$

[3]http://wiki.dbpedia.org/Downloads2015-10
[4]http://yago-knowledge.org

In this expression, $P_c = \bigwedge_{1..n} pc_i(x, A_i) \wedge pc_i(y, A_i)$ corresponds to the condition part of a key expression, and $P_k = \bigwedge_{1..m} pk_i(x, u_i) \wedge pk_i(y, u_i)$ corresponds to the key part. Both AMIE and VICKEY take as input a set of maximal non-keys. These non-keys are obtained from the input dataset using SAKey [25]. Like VICKEY, our adapted variant of AMIE uses the non-keys to restrict the search space by pruning the combinations of properties that do not occur in the non-keys. Unlike VICKEY, AMIE searches exhaustively for all rules that define conditional keys in the input dataset, regardless of their minimality. AMIE therefore requires a post-processing phase where all non-minimal conditional keys are removed. Both AMIE and VICKEY are run with a coverage threshold of 1% on a set of 10 DBpedia classes. We set the confidence threshold of AMIE to 100%, so that VICKEY and the modified AMIE mine exactly the same set of conditional keys for a given dataset.

Results. Our results are shown in Table 2. The first three columns show some statistics about the testing datasets, followed by the number of discovered non-keys (NKs), the runtimes of both VICKEY and AMIE and finally the number of obtained conditional keys (CKs). We observe that a generic rule mining solution cannot handle most of the input datasets in less than 1 day. VICKEY, in contrast, runs on the smaller datasets *Actor*, *Mountain*, *Museum* and *Scientist* in less than 1 hour. For other datasets such as University and Book, VICKEY takes more than 12 hours to mine its conditional keys. In those cases, the system is confronted with a much larger search space, due to a large number of long non-keys (with more than 20 properties). To see why this matters, recall that for each subset of properties co-occuring in a non-key, the search space includes all the combinations of properties in that subset. This number can be prohibitively large for long non-keys. For example, if we consider a non-key with n properties, the graph for a condition on one of the properties will have in the worst case a connected component with $n - 1$ levels, where each level i has $\binom{n-1}{i}$ nodes. Furthermore, this number of combinations has to be explored for each possible condition on the chosen property. Nevertheless, VICKEY can deal with such a large search space by materializing the combinations levelwise. In addition, VICKEY applies the minimality criterion to avoid combinations of properties that are over-specifications of minimal conditional keys. A generic rule mining solution like AMIE cannot count on such optimizations. For example, for the class *Album*, AMIE explores more than 12.3k rules (including intermediate rules), where 6.4k rules correspond to potential conditional keys, while VICKEY explores only 4.1k candidates. This shows that VICKEY's strategy indeed prunes the search space much more effectively.

For the class *city*, VICKEY takes more than two days. This is because every person who ever lived in this city has to be checked for being potentially a condition. In fact, 192 of the 283 properties of the *city* dataset are such inverse properties. If we restrict these relations to the ones that appear in YAGO (as we do in the next section), the mining terminates in less than 15 minutes.

5.2 Conditional keys for entity linking

Setting. We also conducted an extrinsic evaluation, where we show that the conditional keys mined by VICKEY improve the performance of entity linking. This is the task of discovering *sameAs* links between the entities of one KB to the equivalent entities in another KB – under the assumption that both KBs use

Class	#Properties	#Ks	#NKs	#CKs
Actor	16	93	22	748
Album	5	1	2	5864
Book	7	5	2	538
City	17	178	43	5282
Film	9	14	13	26750
Mountain	5	3	2	775
Museum	7	14	5	80
Organisation	17	149	3	9737
Scientist	10	22	8	407
University	9	5	5	449

Table 3: Statistics about linked classes

the same relationships. There are numerous methods for entity linking. Many of them use keys [21, 17], because if some combination of properties is a key, and if an entity in one KB shares these properties with an entity in the other KB, then the two entities must be the same. In our experiment, we show that if the set of keys is augmented with conditional keys, the performance of the entity linking improves.

We wish to emphasize that we are not interested in solving the task of entity linking per se. Rather, we are interested in showing that conditional keys can improve the performance of any method that uses keys for entity linking.

Entity linking. As knowledge bases, we chose DBpedia and YAGO3, because there is a gold standard available for the entity links on the YAGO Web page. We use the same set of classes as for the runtime experiments. We first align the properties of these classes using the ROSA approach [10], and correct the mappings manually. We then rewrite the properties of YAGO using its DBpedia counterparts. The entity linking can be done with the keys mined on DBpedia or with the keys mined on YAGO. We conducted both experiments, and obtained very similar results. Hence, we report here only the former. We run SAKey [25] and VICKEY on DBpedia to find standard and conditional keys, respectively. Table 3 shows the number of common properties, the number of keys (Ks), non-keys (NKs) and conditional keys (CKs) in each DBpedia class. Among others, VICKEY finds that *motto* is a key for universities in Italy and some other countries – but not in all countries; that *wasCreatedInYear* is a key for films by certain actors – but not all actors; and that the name is a key for organizations in certain places – but not all places.

To link the datasets, we use the simple algorithm employed in [25]: For each key, we iterate over the entities in DBpedia that have the key properties. If there is an entity in YAGO that has the same values for these properties, we link the two. For conditional keys, we proceed in the same way, but also check whether the conditions of the key are fulfilled in both datasets.

Class		Recall %	Precision %	F1 %
Actor	Ks	27.43	99.93	**43.05**
	CKs	57.49	99.63	72.91
	Ks+CKs	60.42	99.81	**75.27**
Album	Ks	0.01	100.00	**0.03**
	CKs	15.00	99.39	26.07
	Ks+CKs	15.01	99.39	**26.08**
Book	Ks	3.49	100.00	**6.75**
	CKs	11.31	99.31	20.31
	Ks+CKs	13.33	99.75	**23.51**
City	Ks	61.06	99.98	**75.82**
	CKs	82.55	99.45	90.21
	Ks+CKs	84.70	99.93	**91.69**
Film	Ks	4.06	99.96	**7.80**
	CKs	38.17	96.57	54.72
	Ks+CKs	38.62	97.69	**55.35**
Mountain	Ks	0.22	100.00	**0.44**
	CKs	28.59	99.39	44.41
	K+CKs	28.70	99.39	**44.54**
Museum	Ks	12.05	100.00	**21.51**
	CKs	24.86	100.00	39.82
	Ks+CKs	30.85	100.00	**47.16**
Organisation	Ks	1.10	100.00	**2.17**
	CKs	13.97	98.42	24.46
	K+CKs	14.24	98.63	**24.88**
Scientist	Ks	5.78	98.04	**10.92**
	CKs	16.69	99.93	28.60
	Ks+CKs	19.34	99.41	**32.37**
University	Ks	8.93	99.87	**16.39**
	CKs	22.03	99.14	36.04
	Ks+CKs	25.03	99.55	**40.00**

Table 4: Linking results for YAGO and DBpedia.

Results. Table 4 shows the precision, recall and F1 measure of the entity linking task using a) classic keys, b) conditional keys and c) both types of keys. We first observe that the precision is always over 98%. Conversely, the recall is low in some cases. This happens mainly due to our simple linking method, which uses a strict string equality, and also due to the incompleteness of the data in both YAGO and DBpedia. However, even with this simple method, the use of conditional keys can lead to a significant increase in recall (e.g., from 4% to 38% for the class $Film$) – with a negligible impact on precision. Furthermore, when combining classic keys and conditional keys, the recall improves further, with only minimal impact on precision. Overall, we observe an average increase of 21 percentage points in precision, and of 27.5 points in F1 when both standard keys and conditional keys are used to link the data. The average drop in precision is only 0.5%. This shows that conditional keys can significantly increase the recall of entity linking, while reducing precision only marginally.

6 Conclusion

We have presented VICKEY, an approach to mine conditional keys on RDF datasets. Mining keys is a hard problem due to the size of the search space – and this search space is even larger for conditional keys. Our approach overcomes this problem by restricting the search space to the non-keys found by SAKey [25], and by pruning the search space smartly. This allows VICKEY to mine minimal conditional keys in datasets of up to 4M triples. In an extrinsic evaluation, we have shown that our conditional keys can increase the recall of entity linking by up to 34 percentage points – with negligible impact on precision. The VICKEY system, as well as the datasets and the evaluations, are available at `http://bit.ly/2enJqJo`.

References

[1] M. Al-Bakri, M. Atencia, J. David, S. Lalande, and M. Rousset. Uncertainty-sensitive reasoning for inferring sameas facts in linked data. In *ECAI*, 2016.

[2] M. Atencia, J. David, and F. Scharffe. Keys and pseudo-keys detection for web datasets cleansing and interlinking. In *EKAW*, 2012.

[3] J. Atoum, D. Bader, and A. Awajan. Mining functional dependency from relational databases using equivalent classes and minimal cover. *J. of Computer Science*, 2008.

[4] F. Chiang and R. J. Miller. Discovering data quality rules. In *VLDB*, 2008.

[5] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. In *VLDB*, 2013.

[6] 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 *KDD*, 2014.

[7] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. In *VLDB*, 2014.

[8] W. Fan, Z. Fan, C. Tian, and X. L. Dong. Keys for graphs. In *VLDB*, 2015.

[9] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *TKDE*, 23(5), 2011.

[10] L. Galárraga, N. Preda, and F. Suchanek. Mining Rules to Align Knowledge Bases. In *AKBC workshop*, 2013.

[11] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. *VLDB J.*, 24(6), 2015.

[12] L. A. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.

[13] A. Heise, Jorge-Arnulfo, Quiane-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. In *VLDB*, 2013.

[14] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2), 1999.

[15] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.

[16] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web J.*, 6(2), 2015.

[17] A. Nikolov and E. Motta. Data linking: Capturing and utilising implicit schema-level relations. In *LOD workshop*, 2010.

[18] N. Pernelle, F. Saïs, and D. Symeonidou. An automatic key discovery approach for data linking. *J. of Web Semantics*, 23, 2013.

[19] N. Preda, G. Kasneci, F. M. Suchanek, T. Neumann, W. Yuan, and G. Weikum. Active knowledge: dynamically enriching RDF knowledge bases by web services. In *SIGMOD*, 2010.

[20] S. Razniewski, F. M. Suchanek, and W. Nutt. But What Do We Actually Know? . In *AKBC workshop*, 2016.

[21] F. Saïs, N. Pernelle, and M. Rousset. Combining a logical and a numerical method for data reconciliation. *J. Data Semantics*, 12, 2009.

[22] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. Gordian: efficient and scalable discovery of composite keys. In *VLDB*, 2006.

[23] T. Soru, E. Marx, and A. N. Ngomo. ROCKER: A refinement operator for key discovery. In *WWW*, 2015.

[24] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[25] D. Symeonidou, V. Armant, N. Pernelle, and F. Saïs. SAKey: Scalable Almost Key discovery in RDF data. In *ISWC*, 2014.

[26] D. Symeonidou, I. Sanchez, M. Croitoru, P. Neveu, N. Pernelle, F. Saïs, A. Roland-Vialaret, P. Buche, A. Muljarto, and R. Schneider. Key discovery for numerical data: Application to oenological practices. In *ICCS*, 2016.

[27] D. Vrandečić and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Comm. of the ACM*, 57(10), 2014.

[28] Word Wide Web Consortium. RDF Primer, 2004.

[29] Word Wide Web Consortium. OWL 2 Web Ontology Language, 2012.

[30] C. Wyss, C. Giannella, and E. Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *Data Warehousing and Knowledge Discovery*, 2001.

[31] H. Yao and H. J. Hamilton. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2), 2008.