ROCKER – A Refinement Operator for Key Discovery

Tommaso Soru
Institute of Computer Science,
University of Leipzig
tsoru@informatik.uni-leipzig.de

Edgard Marx
Institute of Computer Science,
University of Leipzig
marx@informatik.uni-leipzig.de

Axel-Cyrille Ngonga Ngomo
Institute of Computer Science,
University of Leipzig
ngonga@informatik.uni-leipzig.de

ABSTRACT

The Linked Data principles provide a decentral approach for publishing structured data in the RDF format on the Web. In contrast to structured data published in relational databases where a key is often provided explicitly, finding a set of properties that allows identifying a resource uniquely is a non-trivial task. Still, finding keys is of central importance for manifold applications such as resource deduplication, link discovery, logical data compression and data integration. In this paper, we address this research gap by specifying a refinement operator, dubbed ROCKER, which we prove to be finite, proper and non-redundant. We combine the theoretical characteristics of this operator with two monotonicities of keys to obtain a time-efficient approach for detecting keys, i.e., sets of properties that describe resources uniquely. We then utilize a hash index to compute the discriminability score efficiently. Therewith, we ensure that our approach can scale to very large knowledge bases. Results show that ROCKER yields more accurate results, has a comparable runtime, and consumes less memory w.r.t. existing state-of-the-art techniques.

Categories and Subject Descriptors

H.4.m [**Information Systems Applications**]: Miscellaneous; I.2.8 [**Computing Methodologies**]: Problem Solving, Control Methods, and Search

Keywords

Semantic Web; Linked Data; link discovery; key discovery; refinement operators

1. INTRODUCTION

The number of facts published in the Linked Data Web has grown considerably over the last years [3]. In particular, large knowledge bases such as LinkedTCGA [20] and LinkedGeoData [24] encompass more than 20 billion triples each. The architectural principles behind the Linked Data Web are akin to those on the Web. In particular, the decentral data publication process leads to facts on the same real-world entities being published across manifold knowledge bases. For example, information on *Austin, Texas* is distributed across several knowledge bases, including DBpedia[1], LinkedGeoData and GeoNames[2]. Given the size of the current Linked Data datasets, providing unique means to characterize resources within existing datasets would facilitate the use of these knowledge bases, for example within the context of entity search, data integration, linked data compression and link discovery [18]. Especially for the link discovery task, being provided with unique descriptions of resources in a knowledge base would allow for the more time-efficient computation of property matchings for link specifications, a task that has been pointed out to be particularly tedious in previous work [4].

In relational databases, keys are commonly either artificial or sets of columns that allow to describe each resource uniquely. Previous works [18, 2, 6] adopt this approach for uniquely describing RDF data and use properties instead of columns. Several problems occur when trying to detect keys for RDF data.

1. Resources from the same datasets might not all have the same properties. For example, in the fragment of DBpedia 3.9 shown in Figure 1, only 50% of the resources have a `:meshNumber`. Thus, while the `:meshNumber` is unique, it cannot be used as a key for this dataset.

2. The inverse problem exists for the `:graySubject`, which covers all resources but is not unique as the trigeminal nerve and the lacrimal nerve have the same `:graySubject`. For our toy dataset, only keys of size larger that 1 exist (e.g., {`:graySubject`, `:grayPage`}).

3. The key discovery problem is exponential in the number of properties n in the knowledge base, as the solution space contains $2^n - 1$ possible sets of keys. Thus, naïve solutions to the key discovery problem do not scale.

Moreover, depending on the use case, key discovery approaches have to be able to detect a single key (e.g., to link resources within a knowledge base) or to detect all keys for a resource (e.g., when integrating data across knowledge bases).

[1] `http://dbpedia.org`
[2] `http://www.geonames.org/`

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741642.

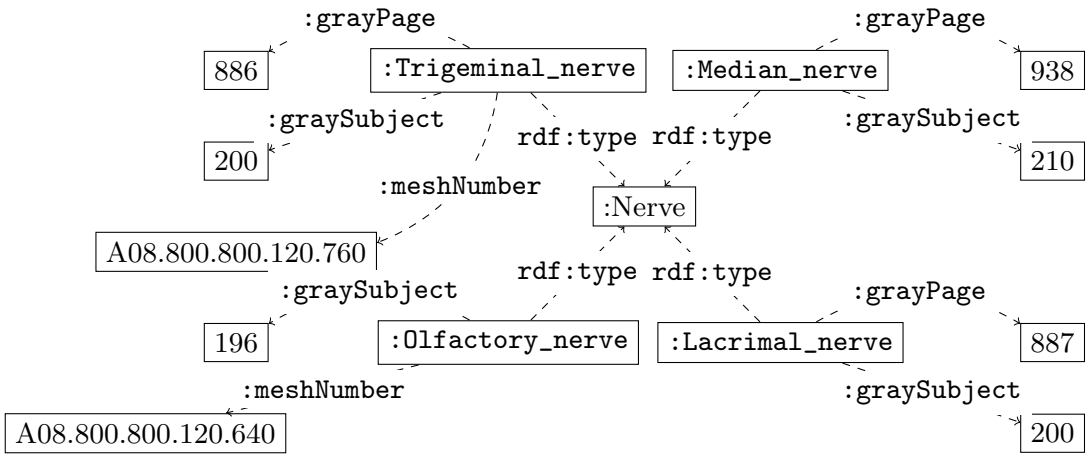


Figure 1: Fragment from a knowledge base on human nerves. The fragment was extracted from DBpedia 3.9.

In this paper, we address the three problems of key discovery within both settings of key discovery (i.e., finding all keys or almost-keys within a given threshold) by using a refinement operator dubbed ρ. This operator is able to detect sets of properties that describe any instance of a given class in a unique manner. By these means, it can generate n-tuples of property values that can be used as keys for resources which instantiate a given class. Our operator relies on a scoring function to compare sets of properties. Based on this comparison, it can efficiently detect single keys, all keys and even predict whether a key can be found in a given dataset. In addition to being finite, non-redundant and proper, our operator also scales well and can thus be used on large knowledge bases. Our contributions are:

- We provide the first refinement operator for key discovery on RDF knowledge bases.
- We prove that our operator is finite, non-redundant, proper, but not complete.
- We utilize the combination of a hash index to compute the discriminability score, i.e. the ability for a set of properties to distinguish their subjects, with two monotonicities of keys to prune the refinement tree and thus ensure that our operator scales.
- We show that our approach succeeds on datasets where current state-of-the-art approaches fail.
- We evaluate our operator on the OAEI instance matching benchmark datasets as well as on DBpedia classes with large populations. In particular, we measure the overall runtime, the memory consumption and the reduction ratio of our approach. Our results suggest that we outperform the state of the art w.r.t. correctness and memory consumption. Moreover, our results suggest that our approach terminates within an acceptable time frame even on very large datasets.

The rest of this paper is structured as follows: We begin by defining the problem at hand formally. Thereafter, we present our operator and prove its theoretical characteristics. After a discussion of related work, we evaluate our operator on synthetic and real data. We then conclude and present some future work.

2. PRELIMINARIES

In the following, we formalize the definition of keys that underlie this paper. This definition is used by our refinement operator to efficiently detect keys.

2.1 Keys

Let K be a finite RDF knowledge base containing instances which belong to a given class and their Concise Bounded Description (CBD).[3] K can be regarded as a set of triples $(s, p, o) \in (\mathcal{R} \cup \mathcal{B}) \times \mathcal{P} \times (\mathcal{R} \cup \mathcal{L} \cup \mathcal{B})$, where $\mathcal{R}$ is the set of all resources, $\mathcal{B}$ is the set of all blank nodes, $\mathcal{P}$ the set of all predicates and $\mathcal{L}$ the set of all literals. We call two resources $r_1, r_2 \in \mathcal{R}$ distinguishable w.r.t. a set of properties $P = \{p_1, \ldots p_n\}$ iff $\exists p \in \{p_1, \ldots p_n\} : \neg((r_1, p, o) \wedge (r_2, p, o))$. Given a knowledge base K, the idea behind *key discovery* is to find one or all sets of properties which make their respective subjects distinguishable in K.

Definition 1 (Key)**.** *We call a set of properties $P \subseteq \mathcal{P}$ a* key *for a knowledge base K (short: key, denoted $key(P, K)$) if all resources in K are distinguishable w.r.t. P.*

Definition 2 (Minimal key)**.** *We call P a* minimal key *(short: mkey) iff P is a key but none of its subsets is. Formally,*

$$mkey(P, K) \Rightarrow key(P, K) \wedge (\neg\exists P' \subset P : key(P', K)). \quad (1)$$

2.2 Discriminability

A *key* for an RDF knowledge base and a *primary composite key* for a database share the same aim. RDF properties represent the projection of database fields into the RDF paradigm, as well as each resource represents a tuple. However, while a tuple element has only one single value, a property might link a resource to more than one RDF objects. Therefore, two resources are distinguishable from each other w.r.t. a set of properties P if their sets of objects are different for at least one $p \in P$.

[3]For the definition of CBD, see `http://www.w3.org/Submission/CBD/`.

To the best of our knowledge, this particular feature of keys was not taken into account by previous works on key discovery for RDF data [18, 6, 2]. For instance, [18, 6] consider two resources r and r' as not distinguishable w.r.t. P if for each $p \in P$ they share at least one object.

Figure 2 shows an example of RDF data, as reported in [6]. Here, the authors claim that $P = \{p_1\} = \{\texttt{:hasActor}\}$ is not a key because `"G.Clooney"` is the object of more than one instance of `:Film`. We would instead consider P as a key, since every film is linked with a different set of objects ($sobj$), i.e.:

$sobj(\texttt{:f1}, p_1) = \{\texttt{"B.Pitt"}, \texttt{"J.Roberts"}\}$
$sobj(\texttt{:f2}, p_1) = \{\texttt{"G.Clooney"}, \texttt{"B.Pitt"}, \texttt{"J.Roberts"}\}$
$sobj(\texttt{:f3}, p_1) = \{\texttt{"B.Pitt"}, \texttt{"G.Clooney"}\}$
$sobj(\texttt{:f4}, p_1) = \{\texttt{"G.Clooney"}, \texttt{"N.Krause"}\}$
$sobj(\texttt{:f5}, p_1) = \{\texttt{"F.Potente"}\}$
$sobj(\texttt{:f6}, p_1) = \varnothing$

Note that `:f6` is still distinguishable from the other resources w.r.t. P, since no other instance of `:Film` in the knowledge base has 0 actors. This particular case was not considered, for example, by the authors of [2].

2.3 Properties of a key

Keys abide by several monotonicities [18]. The first is the so-called *key monotonicity*, which is given by

$$key(P, K) \Rightarrow \forall P' : P \subseteq P' \Rightarrow key(P', K). \quad (2)$$

The reciprocal monotonicity is called the *non-key monotonicity*, which is given by

$$\neg key(P, K) \Rightarrow (\forall P' \subseteq P : \neg key(P', K)). \quad (3)$$

In other words, adding a property to a key yields another key, whilst removing a property to a non-key yields another non-key. In this paper, we present a key discovery approach based on refinement operators.

3. A REFINEMENT OPERATOR FOR KEY DISCOVERY

In this section, we present our refinement operator for key discovery and prove some of its theoretical characteristics. Our formalization is based on that presented in [8].

Let $P \subseteq \mathcal{P}$. Moreover, let $score : 2^{\mathcal{P}} \rightarrow [0, 1]$ be a function that maps each subset P of $\mathcal{P}$ to the fraction of subject resources from K that are distinguishable by using P.

Theorem 1 (Induced quasi-ordering)**.** *The score function induces a quasi-ordering $\preceq$ over the set $\mathcal{P}$, which we define as follows:*

$$P_1 \preceq P_2 \Leftrightarrow \min_{p \in P_1} score(p) \leq \min_{q \in P_2} score(q). \quad (4)$$

The reflexivity and transitivity of $\preceq$ are direct consequences of the reflexivity and transitivity of $\leq$ in $\mathbb{R}$. Note that $\preceq$ is not antisymmetric as two sets of properties P_1 and P_2 can be different and contain the property with the lowest score, leading to $P_1 \preceq P_2$ and $P_2 \preceq P_1$.

Definition 3 (Refinement Operator)**.** *Given a quasi-ordered space (S, op) an upward refinement operator r is a mapping from S to 2^S such that $\forall s \in S : s' \in r(s) \Rightarrow op(s, s')$. s' is then called a generalization of s.*

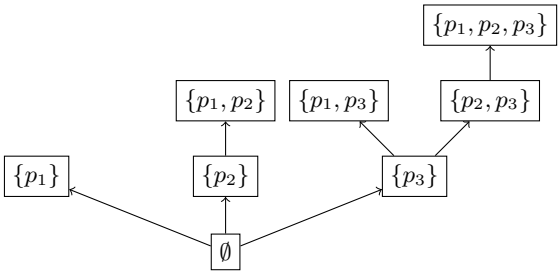


Figure 3: Complete refinement graph for $\mathcal{P} = \{p_1, p_2, p_3\}$. The nodes of the graph are subsets of $\mathcal{P}$. A directed edge (a, b) means $b \in \rho(a)$.

We define our refinement operator over the space $(\mathcal{P}, \preceq)$. First, we begin by ordering the elements of $\mathcal{P}$ according to their score in ascending order, i.e., $\forall p_i, p_j \in \mathcal{P}, i \leq j \Rightarrow score(p_i) \leq score(p_j)$. Then, we can define our operator as follows:

$$\rho(P) = \begin{cases} \mathcal{P} \text{ iff } P = \emptyset, \\ \{P \cup \{p_1\}, \ldots, P \cup \{p_i\}\} \text{ where } p_j \in P \Rightarrow i < j. \end{cases} \quad (5)$$

For example, the complete refinement graph for $\mathcal{P} = \{p_1, p_2, p_3\}$ is given in Figure 3. We use this operator in an iterative manner by only expanding the node with the highest score in the refinement graph. The intuition behind this approach to searching for key is that by ordering properties by their score, we can easily detect and expand the most promising sets of properties without generating redundant nodes. To prove some of the characteristics of ρ, we need to explicate the concept of a refinement chain:

Definition 4 (Refinement chain)**.** *A set $P_2 \in \mathcal{P}$ belong to the refinement chain of $P_1 \in \mathcal{P}$ iff $\exists k \in \mathbb{N} : P_2 \in \rho^k(P_1)$, where* $\rho^k(P) = \begin{cases} \mathcal{P} \text{ iff } k = 0, \\ \rho(\rho^{k-1}(P)) \text{ else .} \end{cases}$

For example, a refinement chain exists between $\{p_3\}$ and $\{p_1, p_2, p_3\}$ in the example shown in Figure 3. There is yet no refinement chain between $\{p_1\}$ and $\{p_2\}$ in the same example.

A refinement operator r over the quasi-ordered space (S, op) can abide by the following criteria.

Definition 5 (Finiteness)**.** *r is finite iff $r(s)$ is finite for all $s \in S$.*

Definition 6 (Properness)**.** *r is proper if $\forall s \in S, s' \in r(s) \Rightarrow s \neq s'$.*

Definition 7 (Completeness)**.** *r is said to be complete if for all s and s', $op(s', s)$ implies that there is a refinement chain between s and s'.*

Definition 8 (Redundancy)**.** *A refinement operator r over the space (S, op) is redundant if two different refinement chains can exist between $s \in S$ and $s' \in S$.*

In the following, we show that ρ is finite, proper and non-redundant but not complete.

Theorem 2. *ρ is finite when applied to a finite knowledge base K.*

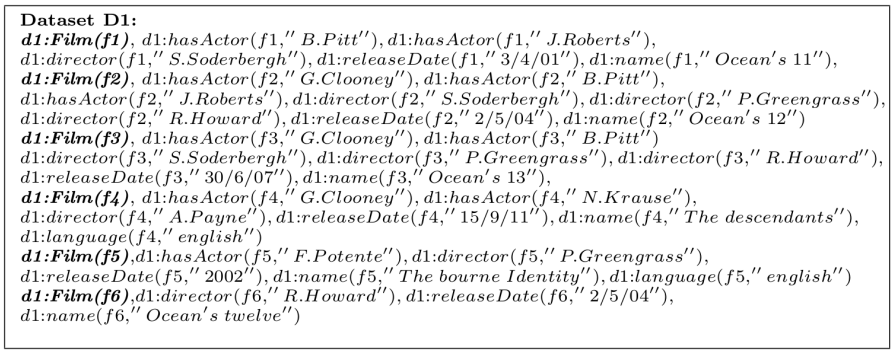


Figure 2: Example of RDF data, as reported in Symeonidou et al., 2014.

Proof. The finiteness of ρ is a direct result of K being finite. The upper bound of the number of properties in K is the number of triples in K. Thus, $|K| < \infty \Rightarrow |\mathcal{P}| < \infty$. Per definition, $|\rho(P)| \leq |\mathcal{P}|$. Thus, we can conclude that $\forall P \in \mathcal{P} : |\rho(P)| < \infty$. □

Theorem 3. *ρ is proper.*

Proof. The properness of ρ also results from the definition of ρ. As we always add exactly a property to P when computing $\rho(P)$, we know that $|\rho(P)| = |P| + 1$. Thus, $\rho(P) \neq P$ must hold. □

Theorem 4. *ρ is not complete.*

Proof. The incompleteness of ρ follows from the definition of $\rho(\emptyset)$. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$. Then $\{p_1\} \preceq \{p_n\}$. Yet, there is clearly no refinement chain between $\{p_n\}$ and $\{p_1\}$ as any subset of $\mathcal{P}$ connected to p_n via a refinement chain must have a magnitude larger than one. Yet, the magnitude of $\{p_1\}$ is 1, which shows that $\{p_1\}$ cannot be connected to $\{p_n\}$ via a refinement chain. □

Theorem 5. *ρ is not redundant.*

Proof. ρ being redundant would mean that a pair of property sets (P, P') exist, where P is linked to P' by two distinct refinement chains C_1 and C_2. Given that these two chains begin and end at the same node, there must be a node N that is common to the two chains but has two distinct fathers N_1 and N_2 that are such that N_1 belongs to C_1 and not to C_2 while N_2 belong to C_2 but not to C_1. Now, N_1 being the father of N means $\exists p_i \in \mathcal{P} : N = N_1 \cup p_i$. Conversely, N_2 being the father of N also means that $\exists p_j \in \mathcal{P} : N = N_2 \cup p_j$. Now if $N_1 \neq N_2$, then we can assume wlog that $i < j$. For $N \in \rho(N_2)$ to hold, j resp. i must be less than the index of any element of N_2 resp. N_1. Moreover, for $N_1 \cup p_i = N_2 \cup p_j$ to hold, p_i would have to already be a element of N_2. However, by virtue of the construction of ρ, this means that $(N_2 \cup \{p_j\}) \notin \rho(N_2)$ given that $p_i \in N_2$ and $i < j$. Thus, we can conclude that there cannot be any to distinct refinement pairs between two subsets of $\mathcal{P}$. □

4. APPROACH

In this section, we present **ROCKER**, a refinement operator approach for key discovery. Our approach was designed with scalability in mind. To this end, it implements a scalable version of the discriminability *score* function based on a hash index. Moreover, we use the monotonicities of keys to check for the existence of keys as well as decide on nodes that need not be refined.

4.1 Implementation

In order to increase the scalability of our operator, we chose an hybrid approach using both in-memory and disk storage database. The following tasks are then performed by ROCKER:

1. the knowledge base model is built using the Apache Jena library;
2. for each class, its instances and properties are extracted;
3. this information is stored and indexed over a *B-tree* structure, whereas the instance URI is used as a key;
4. object values are sorted alphabetically, imploded into a string, indexed using hash codes, and stored into each tuple element;
5. the refinement operator starts from the empty-set node;
6. at each node, the discriminability score is computed by performing a query to the database;
7. the computation terminates according to some rule.

4.2 Definition of the score function

We firstly define the set of subjects of the knowledge base, i.e. the instances of a given class.

$$S = \{s : \exists (s, p, o) \in K\} \tag{6}$$

We then provide the definition of discriminability for two resources w.r.t. a set of properties P.

$$discr(s, s', P) \Leftrightarrow \tag{7}$$

$$\forall s \in S \; \forall p \in P \; \nexists s' \in S : sobj(s, p) \equiv sobj(s', p) \tag{8}$$

where *sobj* is the "set of objects" function introduced in Section 2.2. Finally, we define the score of P (denoted $score(P)$) as the number of subject resources of K that are distinguishable w.r.t. P by using the following formula:

$$score(P) = \frac{|\{s \in S : \forall s' \in S \; s \neq s' \Rightarrow discr(s, s', P)\}|}{|S|}. \tag{9}$$

The set P is a key if $score(P) = 1$, i.e., if P covers all subject resources from K and all resources are distinguishable w.r.t. P.

Basing the refinement on this scoring function has the advantage of allowing ROCKER to cover not only keys but also k-almost-keys [6], which are defined as follows: P is a k-almost-key if $\exists X \subseteq S : |X| \leq k \wedge \forall s, s' \in S \backslash X : discr(s, s', P)$. Consider for example the data shown in Figure 2. If `:f2` did not have `"J. Roberts"` in the list of its actors, then it would not be distinguishable from `:f3`. In this case, the set {`hasActor`} would be 2-almost-key. We can derive a minimal score α for a k-almost-key by simply using the maximal magnitude of X within $score(P)$:

$$|X| \leq k \rightarrow score(P) \geq \frac{|S| - k}{|S|} = \alpha. \quad (10)$$

Note that a key is a 0-almost-key. Moreover, every k-almost-key with $k \geq 0$ is also a $(k+1)$-almost-key.

In our implementation, the score function for P is thus calculated by querying the class table for the columns associated with the properties in P. For each row, the returned values are then concatenated and added to a sorted set, where duplicates are discarded automatically by virtue of the definition of set. The size of this final set represents the numerator for Equation 10.

4.3 Refinement Operator

The pseudo code of ROCKER's refinement operator is shown in Algorithm 1. Given a set of triples K and a set of properties $\mathcal{P}$, we begin by checking whether our ρ-based approach is able to find a key at all. This can be done by computing $score(\mathcal{P})$. If the score is less than 1 (i.e., if $\mathcal{P}$ is not a key), then we know no key exists by virtue of the non-key monotonicity. We can thus terminate and return $\varnothing$, unless a threshold $\alpha < 1$ is given. Under this setting, we terminate if $score(\mathcal{P}) < \alpha$. Now if $\mathcal{P}$ is a key, then some of its subsets might be minimal keys. We then begin by sorting the elements of $\mathcal{P}$ w.r.t. their score. This heuristic tries to make the refinement operator discover keys earlier, so that their descendants can be pruned from the refinement tree, thus decreasing the number of score calculations. A maximal-priority queue is then initialized, where the priority of an element is its score. The queue is initialized with the empty set with a priority of 0. We then take the element P of the queue with the highest priority iteratively and remove it from the queue. Thank to the non-redundancy of ρ, there is no need to check whether P has been seen before. P is refined to P', whose elements p are then checked iteratively. We thus evaluate the scores of all elements of P'. If their score is less than 1, then they are added to the queue. If their score is 1, then we add them to the solution and do not add them to the queue, as they do not need to be refined any further by virtue of the key monotonicity. We then return the set of all keys.

Our approach has several advantages due to the theoretical characteristics of ρ and the key monotonicities:

1. It terminates quickly if there is no key by virtue of using the non-key monotonicity.
2. It is guaranteed to find all existing minkeys by virtue of the key monotonicity.
3. Using a sorted queue, it encourages node pruning by evaluating the most promising nodes first.
4. It never visits the same node twice due to the non-redundancy of ρ.
5. It is ensured to find all existing keys.

Algorithm 1 ROCKER's algorithm for detecting all keys. The algorithm for detecting a single key does not require the solution variable. Instead, it returns the first P having $score(P) = 1$ it finds.

```
Require: Set of triples K
P = {p : ∃s, p with (s, p, o) ∈ K}
if score(P) < 1 then
    return ∅;
end if
P = sortByScore(P);
MaxPriorityQueue q = new Queue();
Set solution = new Set();
q.add(∅, 0); // add ∅ with priority 0
while ¬q.isEmpty() do
    P' = q.getFirst();
    q.removeFirst();
    P = ρ(P');
    for all p ∈ P do
        σ = score(p);
        if σ == 1 then
            solution.add(p);
        else
            q.add(p, σ);
        end if
    end for
end while
return solution;
```

4.4 Search Strategy

As already mentioned in [18], the number of nodes to visit in the key discovery problem is exponential w.r.t. the number of properties considered. More precisely, given n properties, the computational complexity of our algorithm is $O(2^n)$ in the worst case, i.e. when there exists one only key formed by all properties. We tackle this issue by introducing a *fast search* strategy, which can be enabled to speed up the computation. Within this optional setting, whenever a key is found, at the next iteration all branches containing parts of the key are pruned from the refinement tree. This strategy tries to improve the runtime while fostering diversity among the discovered keys. Moreover, we consider properties whose atomic candidate keys have a score greater than a threshold τ. This lets the algorithm discard properties that alone distinguish less instances, thus having a lower probability to be part of a key.

5. RELATED WORK

Key discovery is a rather new research field within the domain of Linked Data, although the issue of finding keys among fields has been inherited from relational databases. However, relational databases do not consider semantics (e.g., subsumption relations) which belong to the core of Linked Data. Previous work on key discovery for the Semantic Web can be found in [18, 2, 6]. For instance, KD2R is an automatic discovery tool for composite keys in RDF data sources that may conform to different schemata [18]. It relies on the creation of prefix trees, which serve for finding maximal undetermined keys and non-keys. However, state-of-the-art

approaches as Linkkey and SAKey have shown to outperform KD2R on runtime and number of generated keys [2, 6]. To the best of our knowledge, not only is ROCKER the first refinement-operator-based approach for key discovery, it is also the first machine-learning-based approach for key discovery.

Independently on the application domain, the key discovery problem is a sub-problem of Functional Dependencies (FDs), as every element is distinguishable only by its attributes. Keys or FDs are widely used in ontology alignment, as well as in data mining [10], reverse engineering [5], and query optimization [9, 7]. In particular, blocking methods such as [12] utilize approximate keys to reduce the computational complexity of dataset joins. Unsupervised learning approaches aim at finding links among datasets by comparing datatype values of properties contained into minimal keys [22]. The so-called collective or global approaches of data linking use keys to generate identity links between instance joins for the final scope of enriching the ontology with the collected information [19, 1].

As previously mentioned, one of the main application areas of ROCKER is link discovery. Several approaches have been developed in previous works to detect matching properties and using them for link discovery. For example, [16] relies on the hospital-residents problem to detect property matches. Other approaches based on genetic programming (e.g., [17]) detect matching properties while learning link specifications, which currently implements several time-efficient approaches for link discovery. [15] proposes an approach based on the Cauchy-Schwarz inequality that allows discarding a large number of superfluous comparisons. HYPPO [13] and $\mathcal{HR}^3$ [14] rely on space tiling in spaces with measures that can be split into independent measures across the dimensions of the problem at hand. In particular, $\mathcal{HR}^3$ was shown to be the first approach that can achieve a relative reduction ratio r' less or equal to any given relative reduction ratio $r > 1$. In the ACIDS approach, similarity measures are performed on property values in order to yield features for machine-learning classifiers as support vectors machines [23]. Amongst other link discovery approaches, RDF-AI [21] relies on a five-step method that comprises the preprocessing, matching, fusion, interlink and post-processing of data sets.

6. EVALUATION

6.1 Experimental Setup

We evaluated ROCKER w.r.t. four characteristics: its runtime, RAM consumption, key extraction quality, and reduction ratio RR [18] between visited and total nodes.

$$RR(\alpha) = 1 - \frac{|vnodes(\alpha)|}{2^{|\mathcal{P}|}}. \quad (11)$$

Our approach was evaluated on data from twelve different datasets. The first two datasets were chosen in order to evaluate ROCKER on an existing artificial benchmark. Both `Restaurant 1` and `2` belong to the Ontology Alignment Evaluation Initiative (OAEI) benchmark. We then evaluated the scalability of ROCKER on ten other datasets generated from DBpedia. We built the datasets using the RDFSlice tool [11], so that each of them contains a class with its instances and their CBD. According to DBtrends[4], these classes rank among the top 20 of the most populated classes in DBpedia 3.9. The domains vary from geography (`Village`, `ArchitecturalStructure`) to professionals (`Artist`, `SoccerPlayer`) and abstract concepts (`PersonFunction`, `CareerStation`).

The generation of new evaluation datasets was preferred over the use of existing datasets due to the following reasons:

1. Datasets from the current state-of-the-art approaches contain a maximum of $1.6M$ triples, while ours scale up to $17.1M$ triples.
2. Some of the existing datasets were not formatted properly.
3. To the best of our knowledge, no key discovery benchmark has been created to date.

The lack of a manually-annotated gold standard for key discovery did not only affect the choice of the datasets. This led us to adopt the number of retrieved keys and the precision to measure the key extraction quality. In fact, while calculating the precision of a key discovery algorithm by annotating the retrieved keys is a feasible task, the set of all minimal keys needs to be known in order to compute the recall.

We compared ROCKER against two state-of-the-art approaches dubbed Linkkey [2] and SAKey [6]. While Linkkey is a tool able to retrieve keys, SAKey is more scalable and able to retrieve also k-almost keys (see Section 4.2).

ROCKER was implemented in Java as part of the link discovery framework LIMES.[5] The datasets and the algorithm source code are also available online.[6] We launched ROCKER with two different settings; the former aims at finding minimal keys ($\alpha = 1$), while the latter aims at finding minimal almost-keys ($\alpha < 1$). Both settings were set to use the *fast search* option with $\tau = 0.001$. For the sake of simplicity, we assigned the same value to α (0.999) for all datasets when retrieving almost-keys. All experiments were carried out on a 64-bit Ubuntu Linux machine with 16 GB of RAM and an octa-core 2.5 GHz CPU.

6.2 Results

Table 1 presents the results we obtained on the twelve datasets. Runtimes in milliseconds are reported for both tasks, i.e. "find minimal keys" and "find minimal almost-keys". For each dataset, the size in number of triples is also shown. As seen in table, all the computation runtimes for the artificial datasets lie within the same magnitude order of 1,000 milliseconds. Both ROCKER runs were slower than the other approaches, however this trend has been disproved by the following results. On the medium-sized datasets `PersonFunction`, `CareerStation` and `OrganisationMember`, our approach is the only one which completed all three tasks. In particular, Linkkey reached the Java heap space on the first two, while SAKey did not complete on the third one. On the seven remaining datasets whose size in NTriples format is larger than 1.5 GB, only our approach was able to finish the computation. This fact leads to consider ROCKER as the most scalable approach for key discovery at the state of the art.

[4] `http://dbtrends.aksw.org/`
[5] `http://limes.sf.net`
[6] `http://github.com/AKSW/rocker/`

As can be read in [2], Linkkey was evaluated on datasets smaller than all the DBpedia datasets we generated. We thus integrated the evaluation carried out by Linkkey's authors by running the tool on our new datasets. At the same time, the largest dataset SAKey was evaluated on is comparable with our medium-sized datasets [6]. Results shown in Table 1 are thus compatible with the evaluations performed by the respective state-of-the-art algorithms.

The node reduction ratio (RR) is shown in Table 2. RR expresses the rate of the number of nodes that were discarded by pruning subtrees, thus avoiding to compute their scores. The number of properties (i.e., the size of $\mathcal{P}$) and the number of visited nodes are also reported.

Table 3 reports the key extraction quality results. For each dataset we show the number of outcomes and the percentage of keys and minimal keys among them (i.e., precision). Many datasets have been omitted as no keys were found by any approach, or simply because the approach failed during the discovery (cf. Table 1). The most interesting results appear on the two OAEI datasets, where Linkkey was not able to recognise any key. On the other hand, SAKey was able to recognise all 3 minimal keys on `Restaurant 2`, yet it returned also 4 non-keys. SAKey was also able to find 2 out of 3 minimal keys, 3 non-minimal keys and 3 non-keys on `Restaurant 1`. Among the other datasets, 3 keys were found on `Village` by ROCKER only.

Table 3: Key extraction quality results.

Dataset	ROCKER	Linkkey	SAKey
Restaurant 1	3 (100%, 100%)	0 (0%, 0%)	8 (62%, 25%)
Restaurant 2	3 (100%, 100%)	0 (0%, 0%)	7 (42%, 42%)
Village	3 (100%, 100%)	-	-

Namespace prefixes and the sets of almost-keys found for `DBpedia ArchitecturalStructure` resp. `DBpedia Village`, using a threshold for the discriminability score $\alpha = 0.999$ are shown below. Reported are 10 almost-keys that were found on `DBpedia ArchitecturalStructure` and the first 20 almost-keys that were found on `DBpedia Village`. As can be seen, the algorithm found six atomic almost-keys. After these had been removed from the maximal-priority queue, the refinement operator followed a path of 109 refinements through the same branch of the refinement tree. Its climb ended on a node having a discriminability score greater than α, as well as a size of 110 properties. After removing this node and its descendants from the queue, the refinement resumed from the bottom of the graph, where ROCKER found 13 more almost-keys composed by 2 properties each. As our algorithm found 84 almost-keys for `DBpedia Village`, the big size of most of the almost-keys may be the reason for the longest computation.

Prefix	Namespace
`dbo:`	http://dbpedia.org/ontology/
`dbp:`	http://dbpedia.org/property/
`dcterms:`	http://purl.org/dc/terms/
`rdfs:`	http://www.w3.org/2000/01/rdf-schema#
`geo:`	http://www.w3.org/2003/01/geo/wgs84_pos#
`foaf:`	http://xmlns.com/foaf/0.1/
`prov:`	http://www.w3.org/ns/prov#

Size	Properties	Score
4	[foaf:name, geo:long, dbo:location, dbp:hasPhotoCollection]	0.99905
4	[foaf:name, geo:long, dbp:hasPhotoCollection, foaf:homepage]	0.99905
4	[foaf:name, geo:long, dbo:elevation, dbp:hasPhotoCollection]	0.99905
4	[foaf:name, geo:long, dbp:hasPhotoCollection, dbo:runwayLength]	0.99905
4	[foaf:name, geo:long, dbo:openingYear, dbp:hasPhotoCollection]	0.99905
4	[dbo:height, foaf:name, geo:long, dbp:hasPhotoCollection]	0.99905
4	[dbo:river, foaf:name, geo:long, dbp:hasPhotoCollection]	0.99905
4	[dbo:buildingStartYear, foaf:name, geo:long, dbp:hasPhotoCollection]	0.99905
4	[foaf:name, geo:long, dbp:hasPhotoCollection, dbo:part]	0.99905
4	[foaf:name, geo:long, dbp:hasPhotoCollection, dbo:primaryFuelType]	0.99905
1	[dbo:wikiPageID]	0.99995
1	[rdfs:label]	0.99995
1	[prov:wasDerivedFrom]	0.99995
1	[dbp:hasPhotoCollection]	0.99995
1	[foaf:isPrimaryTopicOf]	0.99995
1	[dbo:wikiPageRevisionID]	0.99995
110	[geo:lat, dbp:postalCode, dbp:imageFlag, dbp:northeast, ..., dbp:arname]	0.99997
2	[foaf:name, rdfs:comment]	0.99958
2	[geo:long, rdfs:comment]	0.99973
2	[geo:lat, rdfs:comment]	0.99968
2	[rdfs:comment, dbp:name]	0.99955
2	[dbo:wikiPageWikiLink, rdfs:comment]	0.99914
2	[rdfs:comment, dbp:wikiPageUsesTemplate]	0.99911
2	[dbo:isPartOf, rdfs:comment]	0.99901
2	[dbo:wikiPageLength, rdfs:comment]	0.99973
2	[rdfs:comment, dbo:wikiPageExternalLink]	0.99909
2	[rdfs:comment, dcterms:subject]	0.99902
2	[dbp:longd, rdfs:comment]	0.99904
2	[rdfs:comment, dbp:latd]	0.99900
2	[rdfs:comment, dbo:wikiPageOutDegree]	0.99900

7. DISCUSSION

As presented in the previous section, ROCKER improves the state of the art w.r.t. correctness and memory consumption. Other approaches Linkkey and SAkey have shown to require much more memory than ours, as they could not return any result on bigger datasets. In particular, the heap space of 16 GB was reached on 8 and 9 DBpedia datasets, respectively. Unlike the other approaches, ROCKER managed to remain below the heap space by storing the hash index on disk. In fact, in-memory-based algorithms Linkkey and SAKey were not able to handle indexes for datasets having more than 10 million triples.

Runtime results showed that SAKey is the fastest approach on small datasets, being 1.5 to 3 times faster than the others. This could be explained by the fact that the index creation task is quicker for in-memory-based algorithms. Moreover, the outcome analysis presented in Table 3 confirmed that Linkkey and SAKey found candidates that obey their respective key definitions. As mentioned before, the key definition introduced in this work is more correct. A stricter definition leads ROCKER to a farther exploration of the knowledge graph, whereas the other approaches stop. Thus, the runtime is affected. Nevertheless, as can be seen, the runtime is compensated by a substantial improvement in the quality of the results.

In order to analyse how the key discovery task varies w.r.t. the threshold α, we ran ROCKER on one chosen dataset `DBpedia Monument`. Figure 4 shows the number of minimal almost-keys found for values of α within the interval $[0.95, 1]$

Table 1: Runtime results in milliseconds for ROCKER, Linkkey and SAKey on all datasets.

Dataset	Triples	ROCKER(1.0)	ROCKER(0.999)	Linkkey	SAKey
OAEI 2011 Restaurant 1	$1.1K$	1,880	2,170	1,698	1,028
OAEI 2011 Restaurant 2	$7.5K$	2,424	2,833	2,278	885
DBpedia PersonFunction	$383K$	14,565	11,626	OutOfMemory	6,221
DBpedia CareerStation	$3.0M$	79,964	118,632	OutOfMemory	2,199,854
DBpedia OrganisationMember	$3.9M$	1,075,679	1,130,640	227,336	OutOfMemory
DBpedia Album	$11.4M$	1,948,767	366,147	OutOfMemory	OutOfMemory
DBpedia Artist	$12.0M$	203,764	168,049	OutOfMemory	OutOfMemory
DBpedia Village	$12.9M$	4,224,338	18,872,456	OutOfMemory	OutOfMemory
DBpedia Animal	$13.7M$	8,565,772	3,426,372	OutOfMemory	OutOfMemory
DBpedia SoccerPlayer	$13.9M$	314,853	317,285	OutOfMemory	OutOfMemory
DBpedia ArchitecturalStructure	$13.3M$	541,054	1,010,347	OutOfMemory	OutOfMemory
DBpedia MusicalWork	$17.1M$	2,524,120	2,634,869	OutOfMemory	OutOfMemory

Table 2: Reduction ratios for the two settings of ROCKER on all datasets.

Dataset	# properties	vnodes(1.0)	vnodes(0.999)	RR(1.0)	RR(0.999)
OAEI 2011 Restaurant 1	4	6	6	60.00%	60.00%
OAEI 2011 Restaurant 2	4	6	6	60.00%	60.00%
DBpedia PersonFunction	2	3	3	0.00%	0.00%
DBpedia CareerStation	3	4	4	42.86%	42.86%
DBpedia OrganisationMember	20	378	378	99.96%	99.96%
DBpedia Album	103	753	753	∼100.00%	∼100.00%
DBpedia Artist	205	928	928	∼100.00%	∼100.00%
DBpedia Village	116	1387	1700	∼100.00%	∼100.00%
DBpedia Animal	131	1188	1188	∼100.00%	∼100.00%
DBpedia SoccerPlayer	88	528	528	∼100.00%	∼100.00%
DBpedia ArchitecturalStructure	698	1622	3693	∼100.00%	∼100.00%
DBpedia MusicalWork	136	1201	1201	∼100.00%	∼100.00%

with a step of 0.005. As can be seen, values are not in scale, i.e. a minimal almost-key for α_0 does not necessarily belong to the set of minimal almost-keys for $\alpha_1 < \alpha_0$. This is because threshold α can "block" the computation before the following refinement. For instance, the highest value was reported for $\alpha = 0.955$, where 136 minimal almost-keys were found. Most of these keys are formed by a common root of two properties, which we call p_1 and p_2, in the form $\{p_1, p_2, p_i\}$ with $i = 3, \ldots, 96$. Since the discriminability score of $\{p_1, p_2\}$ is 0.953, it is not considered as minimal almost-key for $\alpha = 0.955$. However for $\alpha = 0.95$, $\{p_1, p_2\}$ will be a minimal almost-key and its descendants will not be visited, thus reducing the number of almost-keys and the runtime (see Table 6(a)).

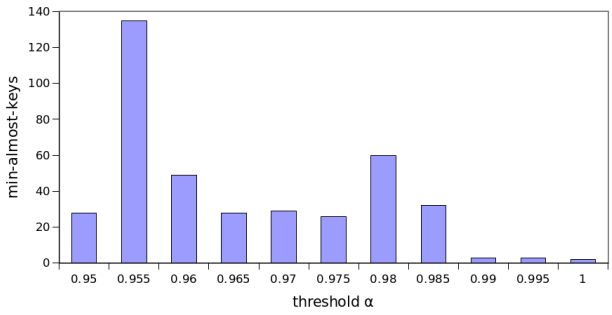


Figure 4: Number of minimal almost-keys found in function of threshold α for ROCKER on dataset `DBpedia Monument`.

Table 6(b) shows the memory consumption w.r.t. α. For $\alpha \geq 0.99$, ROCKER required less memory ($\sim$ 2 GB) than on the other experiments ($\sim$ 5.2 GB), because all the almost-keys were found before visiting the remaining refinement tree. The fact that no other almost-key exists is ensured by evaluating the score for the top element of the refinement tree, which contains all the remaining properties. Having this a score less than α, ROCKER ends the computation.

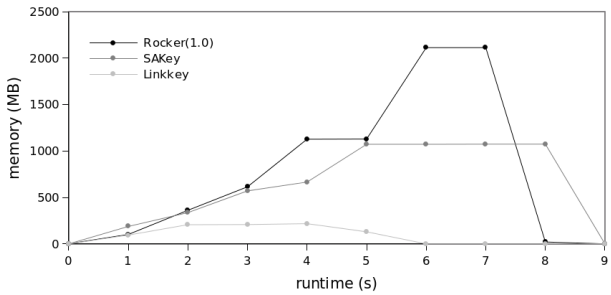
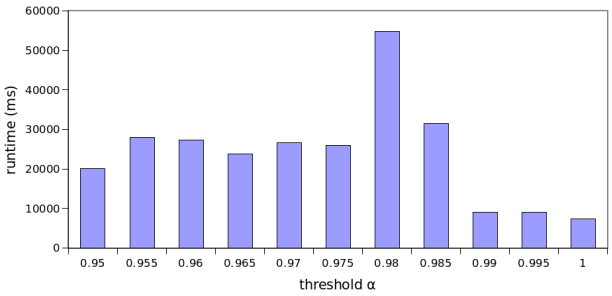


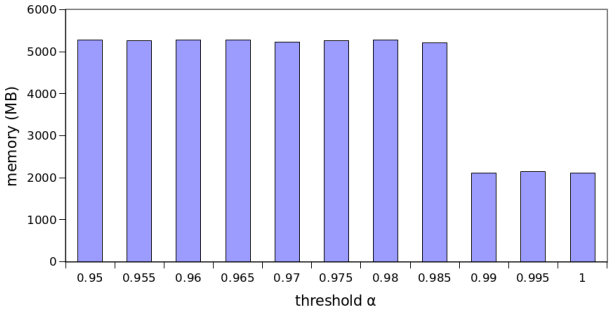
Figure 5: Linkkey showed the best runtime and RAM consumption performances on `DBpedia Monument`, confirming the results in Table 1.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented the first refinement operator for key discovery. We showed that the operator is non-redundant, non-complete and finite. We implemented the operator within the ROCKER approach and showed how it can be extended to scale even on large knowledge bases. Our evaluation of ROCKER suggests that it goes beyond the state of the art with respect to its correctness and memory efficiency, while achieving comparable runtimes. Future directions include a study of the run times, number of keys and visited nodes w.r.t. the input threshold. Then, we will investigate on optimization by using in-memory storage for the hash tables, in order to decrease the query runtimes. Moreover, we will fully integrate the key discovery algorithm in LIMES and make it available in the next releases. We will then experiment with combining key discovery with the



(a)



(b)

Figure 6: Run times (6(a)) and Random Access Memory consumption (6(b)) in function of threshold α for ROCKER on dataset `DBpedia Monument`.

detection of property alignments and use those alignments within the context of link discovery.

9. ACKNOWLEDGMENTS

This research is part of the GeoKnow project, funded by the European Commission with the 7th Framework Programme (Grant Agreement No. 318159).

10. REFERENCES

[1] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 952–963. IEEE, 2009.

[2] M. Atencia, J. David, and J. Euzenat. Data interlinking through robust linkkey extraction. In T. Schaub, G. Friedrich, and B. O'Sullivan, editors, *ECAI*, pages 15–20, 2014.

[3] S. Auer, J. Lehmann, and A.-C. N. Ngomo. Introduction to linked data and its lifecycle on the web. In *Reasoning Web*, pages 1–75, 2011.

[4] M. Cheatham and P. Hitzler. String similarity metrics for ontology alignment. In *The Semantic Web–ISWC 2013*, pages 294–309. Springer, 2013.

[5] R. H. Chiang, T. M. Barron, and V. C. Storey. Reverse engineering of relational databases: Extraction of an eer model from a relational database. *Data & Knowledge Engineering*, 12(2):107–142, 1994.

[6] N. P. Danai Symeonidou, Vincent Armant and F. Saïs. Sakey: Scalable almost key discovery in rdf data. In *ISWC 2014*, 2014.

[7] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658. ACM, 2004.

[8] J. Lehmann and P. Hitzler. Concept learning in description logics using refinement operators. *Machine Learning*, 78(1-2):203–250, 2010.

[9] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies from relations. *Data & Knowledge Engineering*, 12(1):83–99, 1994.

[10] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.*, 1(3):241–258, 1997.

[11] E. Marx, T. Soru, S. Shekarpour, S. Auer, A.-C. Ngonga Ngomo, and K. Breitman. Towards an efficient RDF dataset slicing. *IJSC*, 07(04):455–477, 2013.

[12] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, pages 440–445. AAAI Press, 2006.

[13] A.-C. Ngonga Ngomo. A Time-Efficient Hybrid Approach to Link Discovery. In *OM*, 2011.

[14] A.-C. Ngonga Ngomo. Link Discovery with Guaranteed Reduction Ratio in Affine Spaces with Minkowski Measures. In *ISWC*, pages 378–393, 2012.

[15] A.-C. Ngonga Ngomo and S. Auer. LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In *IJCAI*, pages 2312–2317, 2011.

[16] A.-C. Ngonga Ngomo, J. Lehmann, S. Auer, and K. Höffner. RAVEN – Active Learning of Link Specifications. In *OM*, 2011.

[17] A. Nikolov, M. d'Aquin, and E. Motta. Unsupervised learning of link discovery configuration. In *ESWC*, pages 119–133, 2012.

[18] N. Pernelle, F. Saïs, and D. Symeonidou. An automatic key discovery approach for data linking. *Web Semantics: Science, Services and Agents on the World Wide Web*, 23:16–30, 2013.

[19] F. Saïs, N. Pernelle, and M.-C. Rousset. Combining a logical and a numerical method for data reconciliation. In *Journal on Data Semantics XII*, pages 66–94. Springer, 2009.

[20] M. Saleem, S. S. Padmanabhuni, A.-C. Ngonga Ngomo, J. S. Almeida, S. Decker, and H. F. Deus. Linked cancer genome atlas database. In *ICSC*, pages 129–134. ACM, 2013.

[21] F. Scharffe, Y. Liu, and C. Zhou. Rdf-ai: an architecture for rdf datasets matching, fusion and interlink. In *Proc. IJCAI 2009 workshop on Identity, reference, and knowledge representation (IR-KR), Pasadena (CA US)*, 2009.

[22] D. Song and J. Heflin. Automatically generating data linkages using a domain-independent candidate selection approach. In *ISWC*, pages 649–664. Springer, 2011.

[23] T. Soru and A.-C. Ngonga Ngomo. Active learning of domain-specific distances for link discovery. In *Proceedings of JIST*, 2012.

[24] C. Stadler, J. Lehmann, K. Höffner, and S. Auer. Linkedgeodata: A core for a web of spatial open data. *Semantic Web*, 3(4):333–354, 2012.