CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	SAKey: Scalable almost key discovery in RDF data
Authors	Symeonidou, Danai;Armant, Vincent;Pernelle, Nathalie;Sais, Fatiha
Publication date	2014-10
Original Citation	Symeonidou, D., Armant, V., Pernelle, N. and Sais, F. (2014) "SAKey: Scalable almost key discovery in RDF data", 13th International Semantic Web Conference, ISWC 2014. Riva del Garda, Trento, Italy, 19-23 October, 2014. Springer: The Semantic Web – ISWC 2014, pp. 33-49. DOI: 10.1007/978-3-319-11964-9_3
Type of publication	Conference item
Link to publisher's version	http://link.springer.com/chapter/10.1007/978-3-319-11964-9_3 - 10.1007/978-3-319-11964-9_3
Rights	© 2014 Springer International Publishing. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-11964-9_3 - http://www.springer.com/gp/rights-permissions/obtaining-permissions/882
Download date	2026-11-14 03:39:29
Item downloaded from	https://hdl.handle.net/10468/2471

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

SAKey: Scalable Almost Key discovery in RDF data

Danai Symeonidou[1], Vincent Armant[2], Nathalie Pernelle[1], and Fatiha Saïs[1]

[1] Laboratoire de Recherche en Informatique, University Paris Sud, France,
[2] Insight Center for Data Analytics, University College Cork, Ireland

Abstract. Exploiting identity links among RDF resources allows applications to efficiently integrate data. Keys can be very useful to discover these identity links. A set of properties is considered as a key when its values uniquely identify resources. However, these keys are usually not available. The approaches that attempt to automatically discover keys can easily be overwhelmed by the size of the data and require clean data. We present SAKey, an approach that discovers keys in RDF data in an efficient way. To prune the search space, SAKey exploits characteristics of the data that are dynamically detected during the process. Furthermore, our approach can discover keys in datasets where erroneous data or duplicates exist (i.e., almost keys). The approach has been evaluated on different synthetic and real datasets. The results show both the relevance of almost keys and the efficiency of discovering them.

Keywords: Keys, Identity Links, Data Linking, RDF, OWL2

1 Introduction

Over the last years, the web of data has received a tremendous increase, containing a huge number of RDF triples. Integrating data described in different RDF datasets and creating semantic links among them, has become one of the most important goals of RDF applications. These links express semantic correspondences between ontology entities, or semantic links between data such as *owl:sameAs* links. By comparing the number of resources published on the web with the number of *owl:sameAs* links, the observation is that the goal of building a Web of data is not accomplished yet.

Even if many approaches have been already proposed to automatically discover *owl:sameAs* links (see [6] or [5] for a survey), only some are knowledge-based. In [10, 17, 1], this knowledge can be expressive and specific linking rules can be learnt from samples of data. [14, 11] exploit key constraints, declared by a domain expert, as knowledge for data linking. A key expresses a set of properties whose values uniquely identify every resource of a dataset. Keys can be used as logical rules to clean or link data when a high precision is needed, or to construct more complex similarity functions [14, 7, 17]. Nevertheless, in most of the datasets published on the Web, the keys are not available and it can be difficult, even for an expert, to determine them.

Key discovery approaches have been proposed recently in the setting of the semantic web [3, 12]. [3] discovers *pseudo keys*, keys that do not follow the OWL2 semantics [13] of a key. This type of keys appear to be useful when a local completeness of data is known. [12] discovers OWL2 keys in clean data, when no errors or duplicates exist. However, this approach cannot handle the huge amount of data found on the web.

Data published on the web are usually created automatically, thus may contain erroneous information or duplicates. When these data are exploited to discover keys, relevant keys can be lost. For example, let us consider a "dirty" dataset where two different people share the same social security number (SSN). In this case, SSN will not be considered as a key, since there exist two people sharing the same SSN. Allowing some exceptions can prevent the system from losing keys. Furthermore, the number of keys discovered in a dataset can be few. Even if a set of properties is not a key, it can lead to generate many correct links. For example, in most of the cases the telephone number of a restaurant is a key. Nevertheless, there can be two different restaurants located in the same place sharing phone numbers. In this case, even if this property is not a key, it can be useful in the linking process.

In this paper we present SAKey, an approach that exploits RDF datasets to discover *almost keys* that follow the OWL2 semantics. An almost key represents a set of properties that is not a key due to few exceptions (i.e., resources that do not respect the key constraint). The set of almost keys is derived from the set of non keys found in the data. SAKey can scale on large datasets by applying a number of filtering and pruning techniques that reduce the requirements of time and space. More precisely, our contributions are as follows:

1. the use of a heuristic to discover keys in erroneous data
2. an algorithm for the efficient discovery of *non keys*
3. an algorithm for the efficient derivation of almost keys from non keys

The paper is organized as follows. Section 2 discusses the related works on key discovery and Section 3 presents the data and ontology model. Sections 4 and 5 are the main part of the paper, presenting almost keys and their discovery using SAKey. Section 6 presents our experiments before Section 7 concludes.

2 Related Work

The problem of discovering Functional Dependencies (FD) and keys has been intensively studied in the relational databases field. Key discovery problem can be viewed as a sub-problem of Functional Dependency discovery. Indeed, a FD states that the value of one attribute is uniquely determined by the values of some other attributes. To capture the inherent uncertainty, due to data heterogeneity and data incompleteness, some approaches discover approximate keys and FDs instead of exact keys and FDs only. In [18], the authors propose a way of retrieving non composite probabilistic FDs from a set of data sources. Two strategies are proposed: the first merges the data before discovering FDs, while the second merges the FDs obtained from each data source. In order to find

the approximate FDs that hold in a relation, TANE [8] partitions the tuples into groups based on their attribute values. When the size of the partition is 1, the partition is eliminated based on the fact that its data cannot represent counter-examples of more complex functional dependencies, so the partition is eliminated. Each approximate FD is associated to an error measure which is the minimal fraction of tuples to remove for the FD to hold in the dataset. For the key discovery problem, in relational context, Gordian method [15] allows discovering exact composite keys from relational data represented in a prefix-tree. In order to avoid to scan all the data, the method discovers first the maximal non keys and use them to derive the minimal keys.

In Semantic Web settings where data can be incomplete and may contain multi-valued properties, KD2R [12] aims to derive exact composite keys from a set of non keys discovered on RDF data sources. KD2R, extends [15] to be able to exploit ontologies and consider incomplete data and multivalued properties. Nevertheless, KD2R [12] that is able to discover composite OWL2 keys can be overwhelmed by large datasets and requires clean data. In [3], the authors have developed an approach based on TANE [8] algorithm to discover pseudo-keys (approximate keys) for which a set of few instances may have the same values for the properties of a key. The two approaches [12] and [8] differ on the semantics of the discovered keys in case of identity link computation. Indeed, the first considers the OWL2 [13] semantics, where in the case of multivalued properties, to infer an identity link between two instances, it suffices that these instances share at least one value for each property involved in the key, while in [3], the two instances have to share all the values for each property involved in the key, i.e., local completeness is assumed for all the properties (see [2] for a detailed comparison). In [16], to develop a data linking blocking method, discriminating data type properties (i.e., approximate keys) are discovered from a dataset. These properties are chosen using unsupervised learning techniques and keys of specific size are explored only if there is no smaller key with a high discriminative power. In more details, the aim in [16] is to find the best approximate keys to construct blocks of instances and not to discover the complete set of valid minimal keys that can be used to link data.

Considering the efficiency aspect, different strategies and heuristics can used to optimize either time complexity or space complexity. In both relational or Semantic Web settings, approaches can exploit monotonicity property of keys and the anti-monotonicity property of non keys to optimize the data exploration.

3 Data Model

RDF (Resource Description Framework) is a data model proposed by W3C used to describe statements about web resources. These statements are usually represented as triples $<subject, property, object>$. In this paper, we use a logical notation and represent a statement as $property(subject, object)$.

An RDF data source D can be associated to an ontology which represents the vocabulary that is used to describe the RDF resources. In our work, we consider RDF data sources that conform to OWL2 ontologies. The ontology O

is presented as a tuple ($\mathcal{C}$, $\mathcal{P}$, $\mathcal{A}$) where $\mathcal{C}$ is a set of classes [3], $\mathcal{P}$ is a set of properties and $\mathcal{A}$ is a set of axioms.

In OWL2[4], it is possible to declare that a set of properties is a key for a given class. More precisely, hasKey($CE(ope_1, \ldots, ope_m)$ $(dpe_1, \ldots, dpe_n)$) states that each instance of the class expression CE is uniquely identified by the object property expressions ope_i and the data property expressions dpe_j. This means that there is no couple of distinct instances of CE that shares values for all the object property expressions ope_i and all the data type property expressions dpe_j [5]. The semantics of the construct *owl:hasKey* is defined in [13].

4 Preliminairies

4.1 Keys with exceptions

RDF datasets may contain erroneous data and duplicates. Thus, discovering keys in RDF datasets without taking into account these data characteristics may lead to lose keys. Furthermore, there exist sets of properties that even if they are not keys, due to a small number of shared values, can be useful for data linking or data cleaning. These sets of properties are particularly needed when a class has no keys.

In this paper, we define a new notion of keys with exceptions called n-almost keys. A set of properties is a n-almost key if there exist at most n instances that share values for this set of properties.

To illustrate our approach, we now introduce an example. Fig. 1 contains descriptions of films. Each film can be described by its name, the release date, the language in which it was filmed, the actors and the directors involved.

One can notice that the property $d1$:*hasActor* is not a key for the class $Film$ since there exists at least one actor that plays in several films. Indeed, "*G. Clooney*" plays in films $f2$, $f3$ and $f4$ while "*M. Daemon*" in $f1$, $f2$ and $f3$. Thus, there exist in total four films sharing actors. Considering each film that share actors with other films as an exception, there exist 4 exceptions for the property $d1$:*hasActor*. We consider the property $d1$:*hasActor* as a 4-almost key since it contains at most 4 exceptions.

Formally, the set of exceptions E_P corresponds to the set of instances that share values with at least one instance, for a given set of properties P.

Definition 1. (Exception set). *Let c be a class ($c \in \mathcal{C}$) and P be a set of properties ($P \subseteq \mathcal{P}$). The exception set E_P is defined as:*

$$E_P = \{X \mid \exists Y (X \neq Y) \wedge c(X) \wedge c(Y) \wedge (\bigwedge_{p \in P} \exists U p(X,U) \wedge p(Y,U))\}$$

[3] $c(i)$ will be used to denote that i is an instance of the class c where $c \in \mathcal{C}$

[4] http://www.w3.org/TR/owl2-overview

[5] We consider only the class expressions that represent atomic OWL classes. An *object property expression* is either an *object property* or an inverse *object property*. The only allowed *data type property expression* is a *data type property*.

Dataset D1:
d1:Film(f1), $d1{:}hasActor(f1,'' B.Pitt'')$, $d1{:}hasActor(f1,'' J.Roberts'')$,
$d1{:}director(f1,'' S.Soderbergh'')$, $d1{:}releaseDate(f1,'' 3/4/01'')$, $d1{:}name(f1,'' Ocean's\ 11'')$,
d1:Film(f2), $d1{:}hasActor(f2,'' G.Clooney'')$, $d1{:}hasActor(f2,'' B.Pitt'')$,
$d1{:}hasActor(f2,'' J.Roberts'')$, $d1{:}director(f2,'' S.Soderbergh'')$, $d1{:}director(f2,'' P.Greengrass'')$,
$d1{:}director(f2,'' R.Howard'')$, $d1{:}releaseDate(f2,'' 2/5/04'')$, $d1{:}name(f2,'' Ocean's\ 12'')$
d1:Film(f3), $d1{:}hasActor(f3,'' G.Clooney'')$, $d1{:}hasActor(f3,'' B.Pitt'')$
$d1{:}director(f3,'' S.Soderbergh'')$, $d1{:}director(f3,'' P.Greengrass'')$, $d1{:}director(f3,'' R.Howard'')$,
$d1{:}releaseDate(f3,'' 30/6/07'')$, $d1{:}name(f3,'' Ocean's\ 13'')$,
d1:Film(f4), $d1{:}hasActor(f4,'' G.Clooney'')$, $d1{:}hasActor(f4,'' N.Krause'')$,
$d1{:}director(f4,'' A.Payne'')$, $d1{:}releaseDate(f4,'' 15/9/11'')$, $d1{:}name(f4,'' The\ descendants'')$,
$d1{:}language(f4,'' english'')$
d1:Film(f5),$d1{:}hasActor(f5,'' F.Potente'')$, $d1{:}director(f5,'' P.Greengrass'')$,
$d1{:}releaseDate(f5,'' 2002'')$, $d1{:}name(f5,'' The\ bourne\ Identity'')$, $d1{:}language(f5,'' english'')$
d1:Film(f6),$d1{:}director(f6,'' R.Howard'')$, $d1{:}releaseDate(f6,'' 2/5/04'')$,
$d1{:}name(f6,'' Ocean's\ twelve'')$

Fig. 1: Example of RDF data

For example, in D1 of Fig. 1 we have: $E_{\{d1:hasActor\}} = \{f1, f2, f3, f4\}$, $E_{\{d1:hasActor,\, d1:director\}} = \{f1, f2, f3\}$.
Using the exception set E_P we give the following definition of a n-almost key.

Definition 2. (n-almost key). *Let c be a class ($c \in \mathcal{C}$), P be a set of properties ($P \subseteq \mathcal{P}$) and n an integer. P is a* n-almost key *for c if $|E_P| \leq n$.*

This means that a set of properties is considered as a n-almost key, if there exists from 1 to n exceptions in the dataset. For example, in D1 $\{d1{:}hasActor,\ d1{:}director\}$ is a 3-almost key and also a n-almost key for each $n \geq 3$. By definition, if a set of properties P is a n-almost key, every superset of P is also a n-almost key. We are interested in discovering only minimal n-almost keys, i.e., n-almost keys that do not contain subsets of properties that are n-almost keys for a fixed n.

4.2 Discovery of n-almost Keys from n-non Keys

To check if a set of properties is a n-almost key for a class c in a dataset D, a naive approach would scan all the instances of a class c to verify if at most n instances share values for these properties. Even when a class is described by few properties, the number of candidate n-almost keys can be huge. For example, if we consider a class c that is described by 60 properties and we aim to discover all the n-almost keys that are composed of at most 5 properties, the number of candidate n-almost keys that should be checked will be more than 6 millions. An efficient way to obtain n-almost keys, as already proposed in [15, 12], is to discover first all the sets of properties that are not n-almost keys and use them to derive the n-almost keys. Indeed, to show that a set of properties is not a n-almost key, it is sufficient to find only $(n{+}1)$ instances that share values for this set. We call the sets that are not n-almost keys, n-non keys.

Definition 3. (n-non key). *Let c be a class ($c \in \mathcal{C}$), P be a set of properties ($P \subseteq \mathcal{P}$) and n an integer. P is a* n-non key *for c if $|E_P| \geq n$.*

For example, the set of properties $\{d1{:}hasActor, d1{:}director\}$ is a 3-non key (i.e., there exist at least 3 films sharing actors and directors). Note that, every subset of P is also a n-non key since the dataset also contains n exceptions for this subset. We are interested in discovering only maximal n-non keys, i.e., n-non keys that are not subsets of other n-non keys for a fixed n.

5 The SAKey Approach

The SAKey approach is composed of three main steps: (1) the preprocessing steps that allow avoiding useless computations (2) the discovery of maximal (n+1)-non keys (see Algorithm 1) and finally (3) the derivation of n-almost keys from the set of(n+1)-non keys (see Algorithm 2).

5.1 Preprocessing steps

Initially we represent the data in a structure called *initial map*. In this map, every set corresponds to a group of instances that share one value for a given property. Table 1 shows the initial map of the dataset D1 presented in Fig. 1. For example, the set $\{f2, f3, f4\}$ of $db{:}hasActor$ represents the films that "G.Clooney" has played in.

$d1{:}hasActor$	$\{\{f1, f2, f3\}, \{f2, f3, f4\}, \{f1, f2\}, \{f4\}, \{f5\}, \{f6\}\}$
$d1{:}director$	$\{\{f1,f2,f3\}, \{f2, f3, f5\}, \{f2, f3, f6\}, \{f4\}\}$
$d1{:}releaseDate$	$\{\{f1\}, \{f2, f6\}, \{f3\}, \{f4\}, \{f5\}\}$
$d1{:}language$	$\{\{f4, f5\}\}$
$d1{:}name$	$\{\{f1\}, \{f2\}, \{f3\}, \{f4\}, \{f5\}, \{f6\}\}$

Table 1: Initial map of D1

Data filtering. To improve the scalability of our approach, we introduce two techniques to filter the data of the initial map.
1. Singleton sets filtering. Sets of size 1 represent instances that do not share values with other instances for a given property. These sets cannot lead to the discovery of a n-non key, since n-non keys are based on instances that share values among them. Thus, only sets of instances with size bigger than 1 are kept. Such sets are called *v-exception sets*.

Definition 4. (v-exception set E_p^v). *A set of instances $\{i_1, \ldots, i_k\}$ of the class c is a E_p^v for the property $p \in \mathcal{P}$ and the value v iff $\{p(i_1, v), \ldots, p(i_k, v)\} \subseteq D$ and $|\{i_1, \ldots, i_k\}| > 1$.*

We denote by $\mathfrak{E}_p$ the *collection* of all the v-exception sets of the property p.

$$\mathfrak{E}_p = \{E_p^v\}$$

For example, in D1, the set $\{f1, f2, f3\}$ is a v-exception set of the property $d1{:}director$.

Given a property p, if all the sets of p are of size 1 (i.e., $\mathfrak{E}_p = \emptyset$), this property is a 1-almost key (key with no exceptions). Thus, singleton sets filtering allows

the discovery of single keys (i.e., keys composed from only one property). In D1, we observe that the property $d1{:}name$ is an 1-almost key.

2. v-exception sets filtering. Comparing the n-non keys that can be found by two v-exception sets $E_p^{v_i}$ and $E_p^{v_j}$, where $E_p^{v_i} \subseteq E_p^{v_j}$, we can ensure that the set of n-non keys that can be found using $E_p^{v_i}$, can also be found using $E_p^{v_j}$. To compute all the maximal n-non keys of a dataset, only the maximal v-exception sets are necessary. Thus, all the non maximal v-exception sets are removed. For example, the v-exception set $E_{hasActor}^{\text{“}J.\ Roberts''}$ $\{f1, f2\}$ in the property $hasActor$ represents the set of films in which the actress “*J. Roberts*” has played. Since there exists another actor having participated in more than these two films (i.e., “*B, Pitt*” in films $f1$, $f2$ and $f3$), the v-exception set $\{f1, f2\}$ can be suppressed without affecting the discovery of n-non keys.

Table 2 presents the data after applying the two filtering techniques on the data of table 1. This structure is called *final map*.

$d1{:}hasActor$	$\{\{f1, f2, f3\}, \{f2, f3, f4\}\}$
$d1{:}director$	$\{\{f1,f2,f3\}, \{f2, f3, f5\}, \{f2, f3, f6\}\}$
$d1{:}releaseDate$	$\{\{f2, f6\}\}$
$d1{:}language$	$\{\{f4, f5\}\}$

Table 2: Final map of D1

Elimination of irrelevant set of properties. When the properties are numerous, the number of candidate n-non keys is huge. However, in some cases, some combinations of properties are irrelevant. For example, in the DBpedia dataset, the properties *depth* and *mountainRange* are never used to describe the same instances of the class $NaturalPlace$. Indeed, *depth* is used to describe natural places that are lakes while *mountainRange* natural places that are mountains. Therefore, *depth* and *mountainRange* cannot participate together in a n-non key. In general, if two properties have less than n instances in common, these two properties will never participate together to a n-non key. We denote by *potential n-non key* a set of properties sharing two by two, at least n instances.

Definition 5. (Potential n-non key). *A set of properties $pnk_n = \{p_1, ..., p_m\}$ is a* potential n-non key *for a class c iff:*

$$\forall \{p_i, p_j\} \in (pnk_n \times pnk_n) \mid I(p_i) \cap I(p_j)| \geq n$$

where I(p) is the set of instances that are subject of p.

To discover all the maximal n-non keys in a given dataset it suffices to find the n-non keys contained in the set of maximal potential n-non keys (PNK). For this purpose, we build a graph where each node represents a property and each edge between two nodes denotes the existence of at least n shared instances between these properties. The maximal potential n-non keys correspond to the maximal cliques of this graph. The problem of finding all maximal cliques of a graph is NP-Complete [9]. Thus, we approximate the maximal cliques using a greedy algorithm inspired by the min-fill elimination order [4].

In D1, $PNK = \{\{db{:}hasActor, db{:}director, db{:}releaseDate\},\{db{:}language\}\}$ corresponds to the set of maximal potential n-non keys when n=2. By construction, all the subsets of properties that are not included in these maximal potential n-non keys are not n-non keys.

5.2 n-non keys discovery

We first present the basic principles of the n-non keys discovery. Then, we introduce the pruning techniques that are used by the $nNonKeyfinder$ algorithm. Finally, we present the algorithm and give an illustrative example.

Basic principles. Let us consider the property $d1{:}hasActor$. Since this property contains at least 3 exceptions, it is considered as a 3-non key. Intuitively the set of properties $\{d1{:}hasActor, d1{:}director\}$ is a 3-non key iff there exist at least 3 distinct films, such that each of them share the same actor and director with another film. In our framework, the sets of films sharing the same actor is represented by the collection of v-exception sets $\mathfrak{E}_{hasActor}$, while the sets of films sharing the same director is represented by the collection of v-exception sets $\mathfrak{E}_{director}$. Intersecting each set of films of $\mathfrak{E}_{hasActor}$ with each set of films of $\mathfrak{E}_{director}$ builds a new collection in which each set of films has the same actor and the same director. More formally, we introduce the *intersect operator* $\otimes$ that intersects collections of exception sets only keeping sets greater than one.

Definition 6. (Intersect operator $\otimes$). *Given two collections of v-exception sets $\mathfrak{E}_p$ and $\mathfrak{E}_{p'}$, we define the intersect $\otimes$ as follow:*

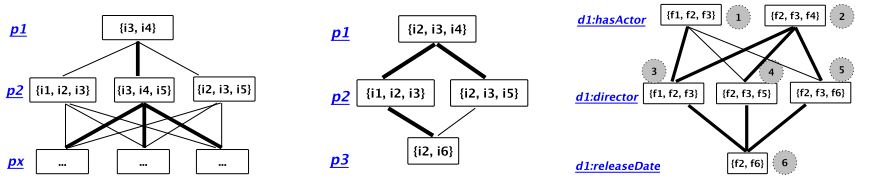
$$\mathfrak{E}_{p_i} \otimes \mathfrak{E}_{p_j} = \{E^v_{p_i} \cap E^v_{p_j} \mid E^v_{p_i} \in \mathfrak{E}_{p_i}, E^v_{p_j} \in \mathfrak{E}_{p_j}, and\ |E^v_{p_i} \cap E^v_{p_j}| > 1\}$$

Given a set properties P, the set of exceptions E_P can be computed by applying the intersect operator to all the collections $\mathfrak{E}_p$ such that $p \in P$.

$$E_P = \bigcup \bigotimes_{p \in P} \mathfrak{E}_p$$

For example for the set of properties $P = \{d1{:}hasActor,\ d1{:}hasDirector\}$, $\mathfrak{E}_P$=$\{\{f_1, f_2, f_3\}, \{f_2, f_3\}\}$ while $E_P = \{f_1, f_2, f_3\}$

Pruning strategies. Computing the intersection of all the collections of v-exception sets represents the worst case scenario of finding maximal n-non keys within a potential n-non key. We have defined several strategies to avoid useless computations. We illustrate the pruning strategies in the Fig. 2 where each level corresponds to the collection $\mathfrak{E}_p$ of a property p and the edges express the intersections that should be computed in the worst case scenario. Thanks to the prunings, only the intersections appearing as highlighted edges are computed.
1. Antimonotonic pruning. This strategy exploits the anti-monotonic characteristic of a n-non key, i.e., if a set of properties is a n-non key, all its subsets



(a) Inclusion Pruning (b) Seen Intersection Pruning (c) example of $nNonKeyFinder$

Fig. 2: $nNonKeyFinder$ prunings and execution

are by definition n-non keys. Thus, no subset of an already discovered n-non key will be explored.

2. Inclusion pruning. In Fig. 2(a) we notice that the v-exception set of p_1 is included in one of the v-exception sets of the property p_2. This means that the biggest intersection between p_1 and p_2 is $\{i_3, i_4\}$. Thus, the other intersections of these two properties will not be computed and only the subpath starting from the v-exception set $\{i_3, i_4, i_5\}$ of p_2 will be explored (bold edges in Fig. 2(a)). Given a set of properties $P = \{p_1, \ldots, p_{j-1}, p_j, \ldots, p_n\}$, when the intersection of $p_1, \ldots, p_{j-1}$ is included in any v-exception set of p_j only this subpath is explored.

3. Seen intersection pruning. In Fig. 2(b) we observe that starting from the v-exception set of the property p_1, the intersection between $\{i_2, i_3, i_4\}$ and $\{i_1, i_2, i_3\}$ or $\{i_2, i_3, i_5\}$ will be in both cases $\{i_2, i_3\}$. Thus, the discovery using the one or the other v-exception set of p_2 will lead to the same n-almost keys. More generally, when a new intersection is included in an already computed intersection, this exploration stops.

***n*NonkeyFinder algorithm.** To discover the maximal n-non keys, the v-exception sets of the final map are explored in a depth-first way. Since the condition for a set of properties P to be a n-non key is $E_P \geq n$ the exploration stops as soon as n exceptions are found.

The algorithm takes as input a property p_i, $curInter$ the current intersection, $curNKey$ the set of already explored properties, $seenInter$ the set of already computed intersections, $nonKeySet$ the set of discovered n-non keys, E the set of exceptions E_P for each explored set of properties P, n the defined number of exceptions and PNK the set of maximal potential n-non keys.

The first call of $nNonKeyFinder$ is: $nNonKeyFinder(p_i, I, \emptyset, \emptyset, \emptyset, \emptyset, n, PNK)$ where p_i is the first property that belongs to at least one potential n-non key and $curInter$ the complete set of instances I. To ensure that a set of properties should be explored, the function $uncheckedNonKeys$ returns the potential n-non keys that (1) contain this set of properties and (2) are not included in an already discovered n-non key in the $nonKeySet$. If the result is not empty, this set of properties is explored. In Line 3, the Inclusion pruning is applied i.e., if the $curInter$ is included in one of the v-exception sets of the property p_i, the $selected\mathfrak{E}_p$ will contain only the $curInter$. Otherwise, all the v-exception sets of

Algorithm 1: *nNonKeyFinder*

```
   Input: p_i, curInter, curNKey, seenInter, nonKeySet, E, n
   Output: nonKeySet: set of the non keys
 1 uncheckedNonKeys ← unchecked({pi} ∪ curNKey, nonKeySet, PNK)
 2 if uncheckedNonKeys ≠ ∅ //PNK and Antimonotonic Pruning then
 3     if (curInter ⊆ E^v_{p_i} s.t. E^v_{p_i} ∈ 𝔈_{p_i}) //Inclusion Pruning then
 4         selected𝔈_{p_i} ← {{curInter}}
 5     else
 6         selected𝔈_{p_i} ← 𝔈_{p_i}
 7     for each E^v_{p_i} ∈ selected𝔈_{p_i} do
 8         newInter ← E^v_{p_i} ∩ curInter
 9         if (|newInter| > 1) then
10             if (newInter ⊈ k s.t. k ∈ seenInter) //Seen Intersection Pruning
               then
11                 nvNKey ← {p_i} ∪ curNKey
12                 update(E, nvNKey, newInter)
13                 if (|E_{nvNkey}| > n) then
14                     nonKeySet ← nonKeySet ∪ {nvNKey}
15                 if ((i + 1) < # properties) then
16                     nNonKeyFinder(p_{i+1}, newInter, nvNKey, seenInter, nonKeySet, E, n)
17         seenInter ← seenInter ∪ {newInter}
18 if ((i + 1) < # properties) then
19     nNonKeyFinder(p_{i+1}, curInter, curNKey, seenInter, nonKeySet, E, n)
```

the property p_i are selected. For each selected v-exception set of the property p_i, all the maximal n-non keys using this v-exception set are discovered. To do so, the current intersection ($curInter$) is intersected with the selected v-exception sets of the property p_i. If the new intersection ($newInter$) is bigger than 1 and has not been seen before (Seen intersection pruning), then $p_i \cup curNonKey$ is stored in $nvNkey$. The instances of $newInter$ are added in E for $nvNkey$ using the update function. If the number of exceptions for a given set of properties is bigger than n, then this set is added to the $nonKeySet$. The algorithm is called with the next property p_{i+1} (Line 16). When the exploration of an intersection ($newInter$) is done, this intersection is added to $SeenInter$. Once, all the n-non keys for the property p_i have been found, $nNonKeyFinder$ is called for the property p_{i+1} with $curInter$ and $curNKey$ (Line 19), forgetting the property p_i in order to explore all the possible combinations of properties.

Table 3 shows the execution of $nNonKeyFinder$ for the example presented in the Fig. 2(c) where $PNK = \{\{d1{:}hasActor, d1{:}director, d1{:}releaseDate\}\}$. We represent the properties in the Table 3 by p_1, p_2, p_3 respectively.

p_i	$selected\mathfrak{E}_p$	E_p^v	$curInter$	$curNkey$	$seenInter$	$nonKeySet$	E
p_1	{1, 2}	1	$\{f1, \ldots, f6\}$	{}	{}	$\{\{p_1\}\}$	$\{(p_1) : (f_1, f_2, f_3)\}$
p_2	{3}	3	$\{f1, f2, f3\}$	$\{p_1\}$	{}	$\{\{p_1\}, \{p_1, p_2\}\}$	$\{(p_1) : (f_1, f_2, f_3)$ $(p_1, p_2) : (f_1, f_2, f_3)\}$
p_3	{6}	6	$\{f1, f2, f3\}$	$\{p_1, p_2\}$	{}	$\{\{p_1\}, \{p_1, p_2\}\}$	$\{(p_1) : (f_1, f_2, f_3)$ $(p_1, p_2) : (f_1, f_2, f_3)\}$
p_3	{6}	6	$\{f1, f2, f3\}$	$\{p_1\}$	$\{\{f_1, f_2, f_3\}\}$	$\{\{p_1\}, \{p_1, p_2\}\}$	$\{(p_1) : (f_1, f_2, f_3)$ $(p_1, p_2) : (f_1, f_2, f_3)\}$
p_1	{1, 2}	2	$\{f1, \ldots, f6\}$	{}	$\{\{f_1, f_2, f_3\}\}$	$\{\{p_1\}, \{p_1, p_2\}\}$	$\{(p_1) : (f_1, f_2, f_3)$ $(p_1, p_2) : (f_1, f_2, f_3)\}$
	...	...	...	...	...	...	...
p_3	{6}	6	$\{f_2, f_3, f_6\}$	$\{p_2\}$	$\{\{f_1, f_2, f_3\}$ $\{f_2, f_3, f_4\}\}$	$\{\{p_1\}, \{p_1, p_2\},$ $\{p_2, p_3\}\}$	$\{(p_1) : (f_1, f_2, f_3)$ $(p_1, p_2) : (f_1, f_2, f_3)\}$ $(p_2, p_3) : (f_2, f_6)\}$

Table 3: $nNonKeyFinder$ execution on the example of Fig. 2(c)

5.3 Key Derivation

In this section we introduce the computation of minimal n-almost keys using maximal (n+1)-non keys. A set of properties is a n-almost key, if it is not equal or included to any maximal (n+1)-non key. Indeed, when all the (n+1)-non keys are discovered, all the sets not found as (n+1)-non keys will have at most n exceptions (n-almost keys).

Both [15] and [12] derive the set of keys by iterating two steps: (1) computing the Cartesian product of complement sets of the discovered non keys and (2) selecting only the minimal sets. The complexity of this algorithm is $\Omega(k^m)$ where k is the number of maximum elements of a complement set and m, the number of complement sets. Deriving keys using this algorithm is very time consuming when the number of properties is big. To avoid useless computations, we propose a new algorithm that derives fast minimal n-almost keys, called $keyDerivation$. In this algorithm, the properties are ordered by their frequencies in the complement sets. At each iteration, the most frequent property is selected and all the n-almost keys involving this property are discovered recursively. For each selected property p, we combine p with the properties of the selected complement sets that do not contain p. Indeed, only complement sets that do not contain this property can lead to the construction of minimal n-almost keys. When all the n-almost keys containing p are discovered, this property is eliminated from every complement set. When at least one complement set is empty, all the n-almost keys have been discovered. If every property has a different frequency in the complement sets, all the n-almost keys found are minimal n-almost keys. In the case where two properties have the same frequency, additional heuristics should be taken into account to avoid computations of non minimal n-almost keys. The worst case complexity of $KeyDerivation$ is $O(k^m)$, when every property in the complement sets is contained in only one complement set.

Let us illustrate the key derivation algorithm throughout and example. Let $\mathcal{P}$= {p_1, p_2, p_3, p_4, p_5} be the set of properties. If the set of maximal n-non keys is {{p_1, p_2, p_3}, {p_1, p_2, p_4}, {p_2, p_5}, {p_3, p_5}}, the set of minimal n-almost keys is {{p_1, p_5}, {p_2, p_3, p_5}, {p_3, p_4}, {p_4, p_5}}. In this example, the complement sets are {{p_1, p_2, p_4}, {p_1, p_3, p_4}, {p_3, p_5}, {p_4, p_5}}. The properties of this ex-

Algorithm 2: *keyDerivation*

```
Input: compSets: set of complement sets
Output: KeySet: set of n-almost keys
1  KeySet ← ∅
2  orderedProperties = getOrderedProperties(compSets)
3  for each p_i ∈ orderedProperties do
4      selectedCompSets ← selectSets(p_i, compSets)
5      if (selectedCompSets == ∅) then
6          KeySet = KeySet ∪ {{p_i}}
7      else
8          KeySet = KeySet ∪ {p_i × keyDerivation(selectedCompSets)}
9      compSets = remove(compSets, p_i)
10     if ( ∃ set ∈ compSet s.t. set == ∅ ) then
11         break
12 return KeySet
```

ample are explored in the following order: $\{p_4, p_1, p_3, p_5, p_2\}$. Starting from the most frequent property, p_4, we calculate all the n-almost keys containing this property. The selected complement set that does not contain this property is $\{p_3, p_5\}$. The property p_4 is combined with every property of this set. The set of n-almost keys is now $\{\{p_3, p_4\}, \{p_4, p_5\}\}$. After the elimination of p_4 the sets are: $\{\{p_1, p_2\}, \{p_1, p_3\}, \{p_5\}, \{p_3, p_5\}\}$. The next property to be explored is p_1. The selected complement sets are $\{p_5\}$ and $\{p_3, p_5\}$. To avoid the discovery of non-minimal n-almost keys, we order the properties of the selected complement sets, according to their frequency (i.e., $\{p_5, p_3\}$). To discover n-almost keys containing p_1 and p_5, we only consider the selected complement sets that do not contain p_5. In this case, no complement set is selected and the key $\{p_1, p_5\}$ is added to the n-almost keys. p_5 is locally suppressed for p_1. Since there is an empty complement set, all the n-almost keys containing p_1 are found and p_1 is removed from the complement sets. Continuing with p_3, the complement sets not containing this property are $\{\{p_2\}, \{p_5\}\}$. In this case, both properties have frequency 1, thus the order will not affect the result. The set of n-almost keys is now $\{\{p_1, p_5\}, \{p_2, p_3, p_5\}, \{p_3, p_4\}, \{p_4, p_5\}\}$. The process terminates since removing p_3 from the complement sets will lead to an empty complement set.

6 Experiments

We evaluated SAKey using 3 groups of experiments. In the first group, we demonstrate the scalability of SAKey thanks to its filtering and pruning techniques. In the second group we compare SAKey with KD2R, the only approach that discovers composite OWL2 keys. The two approaches are compared in two steps. First, we compare the runtimes of their non key discovery algorithms and second, the runtimes of their key derivation algorithms. Finally, we show how n-almost

keys can improve the quality of data linking. The experiments are executed on 3 different datasets, DBpedia[6], YAGO[7] and OAEI 2013[8].

The execution time of each experiment corresponds to the average time of 10 repetitions. In all experiments, the data are stored in a dictionary-encoded map, where each distinct string appearing in a triple is represented by an integer. The experiments have been executed on a single machine with 12GB RAM, processor 2x2.4Ghz, 6-Core Intel Xeon and runs Mac OS X 10.8.

6.1 Scalability of SAKey

SAKey has been executed in every class of DBpedia. Here, we present the scalability on the classes $DB{:}NaturalPlace$, $DB{:}BodyOfWater$ and $DB{:}Lake$ of DBpedia (see Fig. 3(b) for more details) when $n = 1$. We first compare the size of data before and after the filtering steps (see Table 4), and then we run SAKey on the filtered data with and without applying the prunings (see Table 5).

Data filtering experiment. As shown in the Table 4, thanks to the filtering steps, the complete set of n-non keys can be discovered using only a part of the data. We observe that in all the three datasets more than 88% of the sets of instances of the initial map are filtered applying both the singleton filtering and the v-exception set filtering. Note that more than 50% of the properties are suppressed since they are single 1-almost keys (singleton filtering).

class	# Initial sets	# Final sets	# Singleton sets	# E_p^v filtered	Suppressed Prop.
$DB{:}Lake$	57964	4856(8.3%)	50807	2301	78 (54%)
$DB{:}BodyOfWater$	139944	14833(10.5%)	120949	4162	120 (60%)
$DB{:}NaturalPlace$	206323	22584(11%)	177278	6461	131 (60%)

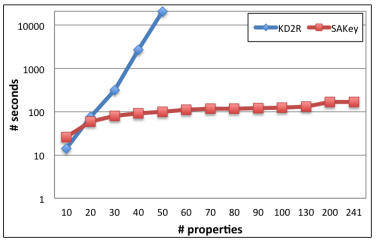
Table 4: Data filtering results on different DBpedia classes

Prunings of SAKey. To validate the importance of our pruning techniques, we run $nNonKeyFinder$ on different datasets with and without prunings. In Table 5, we show that the number of calls of $nNonKeyFinder$ decreases significantly using the prunings. Indeed, in the class $DB{:}Lake$ the number of calls decreases to half. Subsequently, the runtime of SAKey is significantly improved. For example, in the class $DB{:}NaturalPlace$ the time decreases by 23%.

class	without prunings		with prunings	
	Calls	Runtime	Calls	Runtime
$DB{:}Lake$	52337	13s	25289 (48%)	9s
$DB{:}BodyOfWater$	443263	4min28s	153348 (34%)	40s
$DB{:}NaturalPlace$	1286558	5min29s	257056 (20%)	1min15s

Table 5: Pruning results of SAKey on different DBpedia classes

[6] http://wiki.dbpedia.org/Downloads39
[7] http://www.mpi-inf.mpg.de/yago-naga/yago/downloads.html
[8] http://oaei.ontologymatching.org/2013



(a) $nNonKeyFinder$ on $DB{:}NaturalPlace$

class	# triples	# instances	# prop.	KD2R Runtime	SAKey Runtime
$DB{:}Lake$	409016	9438	111	outOfMem.	8s
$DB{:}BodyOfWater$	1068428	34000	200	outOfMem.	37s
$DB{:}NaturalPlace$	1604348	49913	243	outOfMem.	1min10s
$YA{:}Building$	114783	54384	17	26s	9s
$YA{:}SportsSeason$	83944	17839	35	2min	9s
$DB{:}Website$	8506	2870	66	13min	1s
$DB{:}Mountain$	115796	12912	124	191min	11s

(b) $nNonKeyFinder$ in different classes

Fig. 3: $nNonKeyFinder$ runtime for DBpedia and YAGO classes

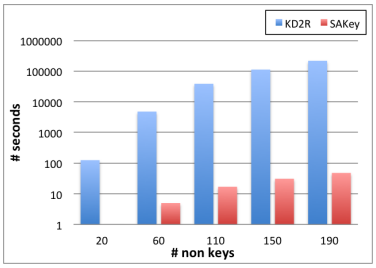
6.2 KD2R vs. SAKey: scalability results

In this section, we compare SAKey with KD2R in two steps. The first experiment compares the efficiency of SAKey against KD2R in the non key discovery process. Given the same set of non keys, the second experiment compares the key discovery approach of KD2R against the one of SAKey. Note that, to obtain the same results from both KD2R and SAKey, the value of n is set to 1.

n-non key discovery. In the Fig. 3(a), we compare the runtimes of the non key discovery of both KD2R and SAKey for the class $DB{:}NaturalPlace$. Starting from the 10 most frequent properties, properties are added until the whole set of properties is explored. We observe that KD2R is not resistant to the number of properties and its runtime increases exponentially. For example, when the 50 most frequent properties are selected, KD2R takes more than five hours to discover the non keys while SAKey takes only two minutes. Moreover, we notice that SAKey is linear in the beginning and almost constant after a certain size of properties. This happens since the class $DB{:}NaturalPlace$ contains many single keys and unlike KD2R, SAKey is able to discover them directly using the singleton sets pruning. In Fig. 3(b) we observe that SAKey is orders of magnitude faster than KD2R in classes of DBpedia and YAGO. Moreover, KD2R runs out of memory in classes containing many properties and triples.

n-almost key derivation. We compare the runtimes of the key derivation of KD2R and SAKey on several sets of non keys. In Fig. 4(a) we present how the time evolves when the number of non keys of the class $DB{:}BodyOfWater$ increases. SAKey scales almost linearly to the number of non keys while the time of KD2R increases significantly. For example, when the number of non keys is 180, KD2R needs more than 1 day to compute the set of minimal keys while SAKey less than 1 minute. Additionally, to show the efficiency of SAKey over KD2R, we compare their runtimes on several datasets (see Fig. 4(b)). In every case, SAKey outperforms KD2R since it discovers fast the set of minimal keys.

In the biggest class of DBpedia, $DB{:}Person$ (more than 8 million triples, 9 hundred thousand instances and 508 properties), SAKey takes 19 hours to compute the n-non keys while KD2R cannot even be applied.



(a) $KeyDerivation$ on $DB{:}BodyOfWater$

Class	# non keys	# keys	KD2R	SAKey
$DB{:}Website$	9	44	1s	1s
$YA{:}Building$	15	34	1s	1s
$YA{:}SportsSeason$	22	175	2s	1s
$DB{:}Lake$	50	480	1min10s	1s
$DB{:}Mountain$	49	821	8min	1s
$DB{:}BodyOfWater$	220	3846	$>$ 1 day	66s
$DB{:}NaturalPlace$	302	7011	$>$ 2 days	5min

(b) $KeyDerivation$ on different classes

Fig. 4: $KeyDerivation$ runtime for DBpedia and YAGO classes

6.3 Data linking with n-almost keys

Here, we evaluate the quality of identity links that can be found using n-almost keys. We have exploited one of the datasets provided by the OAEI'13. The benchmark contains one original file and five test cases. The second file is taken from the first test case. Both files contain DBpedia descriptions of persons and locations (1744 triples, 430 instances, 11 properties). Two resources are linked when they have common values for the all the n-almost key properties (i.e., no similarity measures are used). The recall, precision and f-measure of our linking results has been computed using the gold-standard provided by OAEI'13.

# exceptions	Recall	Precision	F-Measure
0, 1, 2	25.6%	100%	41%
3, 4	47.6%	98.1%	64.2%
5, 6	47.9%	96.3%	63.9%
7, ..., 17	48.1%	96.3%	64.1%
18	49.3%	82.8%	61.8%

Table 6: Data Linking in OAEI 2013

Table 6 shows the obtained results when n varies from 0 to 18. We observe that the quality of the data linking is improved when few exceptions are allowed.

7 Conclusion

In this paper, we present SAKey, an approach for discovering keys on large RDF data under the presence of errors and duplicates. To avoid losing keys when data are "dirty", we discover n-almost keys, keys that are almost valid in a dataset. Our system is able to scale when data are large, in contrast to the state-of the art that discovers composite OWL2 keys. Our extensive experiments show that SAKey can run on millions of triples. The scalability of the approach is validated on different datasets. Moreover, the experiments demonstrate the validity and relevance of the discovered keys.

In our future work, we plan to define a way of automatically set the value of n, in order to ensure the quality of a n-almost key. Allowing no exceptions might be very strict in RDF data while allowing a huge number of exceptions

might end up to many false negatives. We also aim to define a new type of keys, the conditional keys which are keys valid in a subset of the data.

References

1. Arasu, A., Ré, C., Suciu, D.: Large-scale deduplication with constraints using dedupalog. In: ICDE. pp. 952–963 (2009)
2. Atencia, M., Chein, M., Croitoru, M., Jerome David, M.L., Pernelle, N., Saïs, F., Scharffe, F., Symeonidou, D.: Defining key semantics for the rdf datasets: Experiments and evaluations. In: Proceedings of the International Conferences on Conceptual Structures (ICCS)) (2014)
3. Atencia, M., David, J., Scharffe, F.: Keys and pseudo-keys detection for web datasets cleansing and interlinking. In: EKAW. pp. 144–153 (2012)
4. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
5. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. IEEE Transactions on Knowledge and Data Engineering 19, 1–16 (2007)
6. Ferrara, A., Nikolov, A., Scharffe, F.: Data linking for the semantic web. Int. J. Semantic Web Inf. Syst. 7(3), 46–76 (2011)
7. Hu, W., Chen, J., Qu, Y.: A self-training approach for resolving object coreference on the semantic web. In: WWW. pp. 87–96 (2011)
8. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: Tane: An efficient algorithm for discovering functional and approximate dependencies. The Computer Journal 42(2), 100–111 (1999)
9. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations. pp. 85–103. The IBM Research Symposia Series (1972)
10. Low, W.L., Lee, M.L., Ling, T.W.: A knowledge-based approach for duplicate elimination in data cleaning. Information Systems 26, 585–606 (2001)
11. Nikolov, A., Motta, E.: Data linking: Capturing and utilising implicit schema-level relations. In: Proceedings of Linked Data on the Web workshop at 19th International World Wide Web Conference(WWW) (2010)
12. Pernelle, N., Saïs, F., Symeonidou, D.: An automatic key discovery approach for data linking. J. Web Sem. 23, 16–30 (2013)
13. Recommendation, W.: Owl2 web ontology language: Direct semantics. In: Motik B., Patel-Schneider P. F., C.G.B. (ed.) http://www.w3.org/TR/owl2-direct-semantics. W3C (27 October 2009)
14. Saïs, F., Pernelle, N., Rousset, M.C.: Combining a logical and a numerical method for data reconciliation. Journal on Data Semantics 12, 66–94 (2009)
15. Sismanis, Y., Brown, P., Haas, P.J., Reinwald, B.: Gordian: efficient and scalable discovery of composite keys. In: VLDB. pp. 691–702 (2006)
16. Song, D., Heflin, J.: Automatically generating data linkages using a domain-independent candidate selection approach. In: ISWC. pp. 649–664 (2011)
17. Volz, J., Bizer, C., Gaedke, M., Kobilarov, G.: Discovering and maintaining links on the web of data. In: ISWC. pp. 650–665 (2009)
18. Wang, D.Z., Dong, X.L., Sarma, A.D., Franklin, M.J., Halevy, A.Y.: Functional dependency generation and applications in pay-as-you-go data integration systems. In: WebDB (2009)