KD2R: a Key Discovery method for semantic Reference Reconciliation in OWL

Danai Symeonidou

Stage de Master 2 Recherche en Informatique

Greek-French Postgraduate Program, University Paris Sud XI, University of Crete

KD2R: a Key Discovery method for semantic Reference Reconciliation in OWL

Author:
Danai Symeonidou

Supervisors:
Dr. Nathalie Pernelle
Dr. Fatiha Saïs

Host institution:
IASI/LEO Team, INRIA Saclay (Île-de-France) and LRI (Laboratoire de Recherche en Informatique)

Secretary - tel : 01 69 15 75 18 Fax : 01 69 15 42 72
email : Jacques.Laurent@lri.fr

15 March – 9 September 2011

University of Crete, Computer Science Department
University of Paris-Sud XI, Computer Science Department
Leo team, INRIA Saclay and LRI

KD2R : a Key Discovery method for semantic Reference Reconciliation in OWL

A thesis submitted by
Danai Symeonidou
in partial fulfilment for the degree of
Master of Science

Author :

Supervisor :

Committee :

Paris, September 2011

Contents

Summary

The reference reconciliation problem consists of deciding whether different identifiers refer to the same world entity. Some existing reference reconciliation approaches use key constraints to infer reconciliation decisions. In the context of the Linked Open Data, this knowledge is not available. In this master thesis we propose KD2R, a method which allows automatic discovery of key constraints associated to OWL2 classes. These keys are discovered from RDF data which can be incomplete. The proposed algorithm allows this discovery without having to scan all the data. KD2R has been tested on data sets of the international contest OAEI and obtains promising results.

Keywords

Semantic Web, Reference reconciliation, Key discovery, Key Constraints, RDF, OWL

ΠΕΡΙΛΗΨΗ

Ο κύριος άξονας του προβλήματος της συμφιλίωσης αναφορών είναι η ικανότητα ανίχνευσης ότι δυο διαφορετικά αναγνωριστικά αναφέρονται στην ίδια οντότητα. Ορισμένες μέθοδοι που προσπαθούν να λύσουν το πρόβλημα αυτό χρησιμοποιούν κλειδιά για να πάρουν αυτές τις αποφάσεις συμφιλίωσης. Στο πλαίσιο του Linked Open Data τέτοιου είδους πληροφορίες δεν είναι διαθέσιμες. Σε αυτήν την μεταπτυχιακή εργασία προτείνουμε την μέθοδο KD2R, μια μέθοδο που επιτρέπει την αυτόματη ανίχνευση κλειδιών συσχετιζόμενα με OWL2 κλάσεις. Αυτά τα κλειδιά ανιχνεύονται σε RDF αρχεία που μπορεί να είναι ημιτελή. Ο προτεινόμενος αλγόριθμος επιτρέπει την εύρεση κλειδιών χωρίς να είναι απαραίτητη η σάρωση όλων των δεδομένων. Η μέθοδος KD2R έχει δοκιμαστεί σε δεδομένα απο τον διεθνή διαγωνισμό OAEI και τα αποτέλεσματά του είναι πολλά υποσχόμενα.

Keywords

Σημασιολογικός ιστός, συμφιλίωση αναφορών, ανακάλυψη κλειδιών, , RDF, OWL

Resume

Le problème de réconciliation de référence consiste à décider si des identifiants différents référé à la même entité du monde réel. Certaines approches de réconciliation de référence utilisent des contraintes des clé pour déduire des décisions de réconciliation des références. Dans le contexte des données liées, cette connaissance n'est pas disponible. Dans ce stage de master nous proposons KD2R, une méthode qui permet la découverte automatique des contraintes de clé associées à des classes OWL2. Cettes contraintes de clé sont découvertes à partir de données RDF qui peuvent être incomplètes. L'algorithme proposé permet cette découverte, sans avoir à passer en revue toutes les données. KD2R a été testé sur des jeux de données du concours international OAEI et obtient des résultats prometteurs.

Keywords

Web sémantique, la réconciliation de référence, la découverte de clés, les principales contraintes, RDF, OWL

Chapter 1

Introduction

More and more RDF datasets are available in the web. To combine data descriptions coming from different datasets there has to be a method that identifies which descriptions refer to the same objects. The reference reconciliation problem consists of deciding whether different references refer to the same world entity (e.g. the same restaurant, the same gene, etc.). There are a lot of approaches (see [4] or [12] for a survey) that aim to reconcile data. Some existing reference reconciliation approaches use key constraints to infer reconciliation decisions.
The Linked Open Data(LOD) is a cloud in the Web where RDF sources are stored. The Linking Open Data community aims to publish open RDF datasets on the Web and RDF links between data items from different data sources (http://linkeddata.org/home). More than 200 datasets belongs to the LOD cloud including Wordnet, DBpedia or MusicBrainz. In the context of the Linked Open Data, the knowledge of key constraints is not available. This means that information about key constraints is usually missing or we might have only a subset of the real existing keys.

If we were able to find all the combinations of properties that uniquely identify an entity, the reference reconciliation process would be much easier. These properties are in fact the keys. These properties will have a bigger importance in the reconciliation process. This notion of key constraint cannot only be used in RDF but also in databases or even in XML. Some approaches (see [12]) learn property importance on datasets labelled as reconciled. However such datasets are not always available.

In this master thesis, we propose a method for automatic discovery of key constraints. We present KD2R [2] which is an approach for automatic key discovery in RDF data sources which conform to the same (or aligned) OWL ontology. Comparing to existing approaches we do not assume the availability of manually labelled datasets. Nevertheless, to discover keys we consider data sources where the UNA (Unique Name Assumption) is fulfilled. Furthermore, data that are published on the LOD are often partially described regarding to a domain ontology. KD2R is able to discover keys in such incomplete data sources. The Unique Name Assumption (UNA) declares that all the references that appear in a source cannot be reconciled. This means that they refer to different real world entities.

In this work, we propose ways to eliminate the calculations as much as possible since the key discovery can be a really time consuming process when all the data have to be examined. To avoid scanning the whole data source, KD2R [2] discovers first maximal non keys before inferring the keys. KD2R exploits key inheritance between classes in order to prune the non key search space. KD2R approach has been implemented and evaluated on two different data sources.

The report is organized as follows: in section 2, we describe the data and the ontology model and we present how key constraints can be used in the reference reconciliation process. In section 3, we present KD2R and then we present first experiment results in section 4. Finally, in section 6 we conclude and give some future work.

CHAPTER 2

Reference Reconciliation based on key constraints

Before describing in detail how the key constraints can be used, we first present the ontology and the data model that we consider.

2.1 Ontology and Data Model

Data are represented in RDF–Resource Description Framework– (www.w3.org/RDF). For example, the RDF source S1 contains the RDF descriptions of four museums in the form of a set of class facts and property facts (relational notation):

Source S1:
$ArchaeologicalMuseum(S1\_m1), museumName(S1\_m1, Archaeological\ Museum),$
$located(S1\_m1, S1\_c1), museumAddress(S1\_m1, 44\ Patission\ Street),$
$inCountry(S1\_m1, Greece), Museum(S1\_m2), museumName(S1\_m2,$
$Centre\ Pompidou), contains(S1\_m2, S1_p4), contains(S1\_m1, S1\_p5),$
$museumAddress(S1\_m2, 19\ rue\ Beaubourg), inCountry(S1\_m2, France),$
$Museum(S1\_m3), museumName(S1\_m3, Musee\ d'orsay),$
$museumAddress(S1\_m3, 62\ rue\ de\ Lille), inCountry(S1\_m3, France)$
$WaxMuseum(S1\_m4), museumName(S1\_m4, Madame\ Tussauds), located(S1\_m4,$
$S1\_c4), museumAddress(S1\_m4, Marylebone\ Road),\ inCountry(S1\_m4, England)$

The examined RDF data are in conformity with a domain Ontology represented in OWL2 (http://www.w3.org/TR/owl2-overview) The OWL2 Web Ontology Language provides classes, (data or object) properties, individuals and data values. In the Museum ontology (see Figure 2.1), the class $Museum$ is described by its address (*owl:DataProperty* $museumAddress$), its location (*owl:ObjectProperty located*), its name (*owl:DataProperty* $museumName$) and its country (*owl:DataProperty* $inCountry$). The classes $ArcheologicalMuseum$ and $WaxMuseum$ are more specific classes of the class $Museum$.

In OWL2, it is possible to express key axioms for a given class: a key axiom HasKey(CE (OPE1 ... OPEm) (DPE1 ... DPEn)) states that each (named) instance of the class expression CE is uniquely identified by the object property expressions OPEi and by the data property expressions DPEj.[1] This means that no two distinct (named) instances of CE can coincide on the values of all object property expressions OPEi and all data property expressions DPEj. An ObjectPropertyExpression is either an ObjectProperty or InverseObjectProperty. A data property expression is an *owl:DataProperty*.

For example, we can express that the object property $\{located\}$ is a key for the class $City$ using $HasKey(kd2r : City(inverse(kd2r : located))())$. Also the combination of the object property $located$ and the Datatype $museumAddress$ is a key for the class $museum$. This key can be described as: $HasKey(kd2r : Museum((kd2r : located)(kd2r : museumAddress))())$

[1] The ontology can be represented in RDFS or in OWL. In that case, the key axioms can be represented using SWRL(Semantic Web Rule Language) rules.

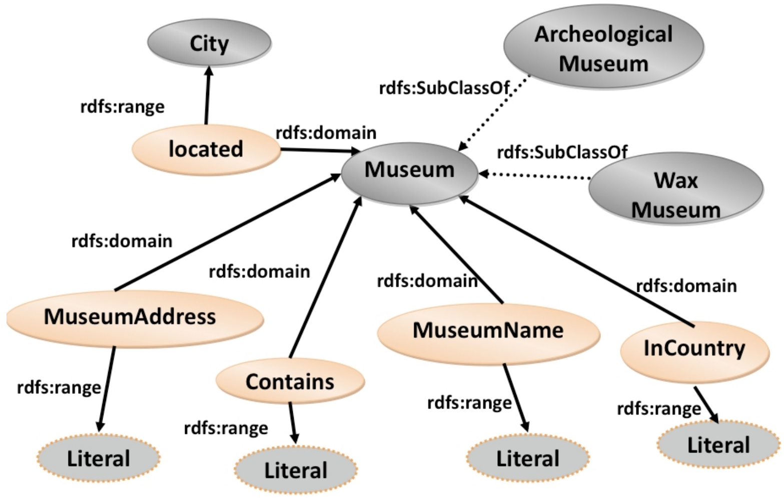

Figure 2.1: Museum Ontology

2.2 Constraint integration in Reference Reconciliation

LN2R [9] is a logical (L2R) and a numerical (N2R) method for reference reconciliation. L2R and N2R use the knowledge given in a OWL (or OWL2) ontology to reconcile data. L2R translates keys, disjunctions between classes and the Unique Name Assumption (UNA) into reconciliation rules.

Unique Name Assumption(UNA): Two different references refer to two distinct entities in the real world.

For example, in a source that describes people, two different references cannot refer to the same person. These rules infer both (non) reconciliation and synonymy facts for literal values. For example, since *located* is a key for the *City* class (one museum is located in only one city) the following rule is generated by L2R:

$$City(L1) \wedge City(L2) \wedge Reconcile(X,Y) \wedge Located(X,L1) \wedge Located(Y,L2) \implies Reconcile(L1,L2)$$

A logical reasoning based on the unit-resolution inference rule is used to infer all the (non) reconciliations.

N2R exploits the ontology in order to generate a similarity function that computes similarity scores for pairs of references. This numerical approach is based on equations that model the influence between similarities. In these equations, each variable represents the (unknown) similarity between two references while the similarities between values of data properties are constants (obtained using standard similarity measures on strings or on sets of strings). Furthermore, ontology and data knowledge (disjunction, UNA) is exploited by N2R in a filtering step to reduce the number of reference pairs that are considered in the equation system. The functions modelling the influence between similarities are a combination of the maximum and the average functions in order to take into account the keys declared in the OWL ontology in an appropriate way (see [9] for more details).

2.3 Key discovery Problem Statement

When RDF data are numerous, heterogeneous and published in clouds of data, the keys that are needed for the reconciliation step are not often available and cannot be easily specified by a human expert. Therefore, we need methods to discover them automatically from data. The key discovery has to face several kinds of problems, due to data heterogeneity: absence of UNA, syntactic variations in data, erroneous values and incompleteness of information.

When UNA is not fulfilled, we cannot distinguish between the two cases:

1. two equal property values describing two references which refer to the same real world entity and

2. two equal property values describing two references which refer to two distinct real world entities.

This ambiguity may lead to missing keys that can be discovered.

The other parameter that affects the key discovery problem is the syntactic variations that may exist in data. This means that the same information can be presented in many different ways and different information can be presented in the same way. This syntactic variation leads to the discovery of incorrect keys.

When sources are extracted from the Web it is possible to find incorrect information (erroneous values) or obsolete data (related to data freshness). This case makes the key discovery more difficult, since this can lead us to discover keys that are likely to be wrong and also lose some real keys.

In RDF data each instance of a class can be described by a subset of properties that are declared in the ontology. The incompleteness of data entails the discovery of keys that may be incorrect.

In this master thesis we focus on the problem of key discovery in incomplete RDF data when UNA assumption is declared for each data source and where there are no erroneous values. We assume also that the data have been normalized and there are no syntactic variations.

Chapter 3

KD2R: Key Discovery method for Reference Reconciliation

KD2R [2] method aims to discover keys as exact as possible, with respect to a given dataset in order to enrich a possible existing key set. These keys define the sets of properties that have a strong influence on the similarity of references as it is done in LN2R method.

The most naive automatic way to discover the keys is to check all the possible combinations of properties that refer to a class. The keys should uniquely identify each instance of a class. Let us assume that we have a class which is described by 15 properties in order to estimate the cost of this naive way. In this case the number of candidate keys is $2^{15} - 1$. In order to minimize the number of computations as much as possible we have proposed a method inspired from [10] which first retrieves the set of maximal non keys and then computes the set of minimal keys, using this set of non keys. Indeed, to make sure that a set of properties is a key we have to scan the whole set of instances of a given class. On the contrary, finding two instances that share the same values for the considered set of properties would suffice to be sure that this set of properties is a non-key. Since RDF data might be incomplete, we introduce the notion of undetermined keys which cannot be considered either as keys or as non keys. Distinguishing undetermined keys from keys will:

1. help a human expert in the validation process of key constraints
2. be used differently in the reconciliation process

We present, first, how we have defined non keys, keys and undetermined keys for a class in a given RDF data source and for a given set of RDF data sources. Then we will present the KD2R-algorithm that is used to find keys for the ontology classes.

3.1 Keys, Non Keys and Undetermined Keys

Let S be a data source for which the UNA is declared, and P_c be the set of RDF properties defined for a class C of the ontology O.

Definition 1 (Non keys).

A set of property expressions $nk_{CSi} = \{pe_1, \ldots, pe_n\}$ is a non key for the class C in S if:

$\exists X \in S, \exists Y \in S$ s.t. $(C(X) \wedge C(Y) \wedge pe_1(X, a_1)) \wedge pe_1(Y, a_1) \wedge \ldots \wedge pe_n(X, a_n)) \wedge pe_n(Y, a_n)) \wedge X \neq Y)$

We denote NK_{CS} the set of non keys $\{nk_{CS1}, ..., nk_{CSm}\}$ of the class C, w.r.t, the data source S.

For example,$\{InCountry\}$ is a key for the class *museum* since there are two museums that are in the same country (Pompidou and Musee d'Orsay are both located in Paris).

Definition 2 (Keys).

A set of property expressions $k_{CSi} = \{pe_1, \ldots, pe_n\}$ is a key for the class C in S if:

$\forall X \in S, \forall Y \in S\ (C(X) \wedge C(Y)) \rightarrow (\exists pe_j \in k_{CSi}$ s.t. $pe_j(X, a)) \wedge pe_j(Y, b)) \wedge a \neq b)$

We denote K_{CS} the key set $\{k_{CS1}, ..., k_{CSm2}\}$ of the class C w.r.t the data source S.

For example, $\{MuseumAddress\}$ is a k_{CS} since the addresses of all the museums that appear in the source are distinct. Each address uniquely identifies a museum in the source.

Definition 3 (Undetermined Keys).

A set of property expressions $uk_{CSi} = \{pe_1, \dots, pe_n\}$ is an undetermined key for the class C in S if: (i) $uk_{CSi} \notin NK_{CS}$ and (ii) $\exists X \in S, \exists Y \in S$ s.t. $((C(X) \wedge C(Y) \wedge \forall pe_j \in uk_{CSi}(pe_j(X,a) \wedge pe_j(Y,b) \implies a = b) \wedge \exists pe_w \in uk_{CSi}$ s.t.$(\nexists pe_w(X,Z) \vee \nexists pe_w(Y,V))$

For example, $\{InCountry, Located\}$ is an undetermined key, since there are two museums in the same country but one of the cities is unknown. Hence, we cannot decide if it represents a key or a non-key.
We denote UK_{CS} the set of keys $\{uk_{CS1}, ..., uk_{CSm3}\}$ of the class C w.r.t the data source S.

Definition of maximal non and undetermined key:
A non key (or a undetermined key respectively) is considered as a maximal non key (or a undetermined key) if it doesn't exist a bigger superset of this non key (or undetermined key) that is also a non key(or a undetermined key).
Example: If for example $\{inCountry, located, contains\}$ is a maximal undetermined key for the of the RDF data described in section 2 ,$\{inCountry, located\}$ can be also an undetermined key but not a maximal one, since it is a subset of a bigger undetermined key

Definition of minimal keys:
A key is considered as a minimal key if it doesn't exist a smaller subset of this key that is also a key. More specifically, a minimal key is the smallest key that we can obtain.
Example: If for example $\{MuseumAddress\}$ is a minimal key for the of the RDF data described in section 2 ,$\{MuseumAddress, located\}$ or $\{MuseumAddress, inCountry\}$ can be also key but not minimal ones, since they are a superset of a smaller key

Keys for a given set of data sources.

Let $\mathcal{S} = \{S1, S2, \dots, Sm\}$ be a set of m data sources for which the UNA is declared. Let $K_{CS1}, \dots, K_{CSm}$ be the respective set of keys of $S1, S2, \dots, Sm$, the set of keys $Kc_{\mathcal{S}}$ that is satisfied in all the sources is the set of minimal keys that belong to the Cartesian product of $K_{CS1}, \dots, K_{CSm}$.

3.2 KD2R Algorithm

Given a set of datasets and a domain ontology, KD2R-algorithm [2] allows to find keys for each instantiated class. It follows a top-down computation in the sense that the keys that are discovered for a class are inherited by its sub-classes. KD2R uses a compact representation of RDF data expressed in a prefix-tree in order to compute the complete set of maximal undermined keys and maximal non keys and then the complete set of minimal keys.

3.2.1 Prefix-Tree creation.

In this section we will present the creation of the prefix-tree which represents the RDF descriptions of a given class.

As it is illustrated in Figure 3.2, each level of the tree corresponds to an instantiated property expression. Each node contains a variable number of cells. Each cell contains:

1. a property value or a distinct URI of the object property expression of the considered level

2. an attribute that records if the value of the cell is null or not
3. a list of URIs referring to the corresponding class instances

Each non-leaf cell has a pointer to a single child node. Each prefix path represents the set of instances that share one value[1] or one URI for all the properties involved in the path.

In order to represent the cases where property values are not given (i.e. null values in relational databases) we create first an intermediate prefix-tree. In this intermediate prefix-tree, an artificial null value is created for those properties. Then, the final prefix-tree is generated by assigning the set of all the possible values to each artificial null value, i.e. those existing in the dataset.

3.2.2 Intermediate Prefix-Tree creation.

In order to create the intermediate prefix-tree, we use the set of all properties that appear at least in one instance of the considered class. For each instance, for each property and for each value if there is no cell which already contains the property value a new cell is created. Otherwise, the cell is updated by adding the instance URI to its associated list of URIs. When a property does not appear in the source, we create or update, in the same way, a cell with an artificial null value. In this case there is an attribute that records that the cell is null. Let it be noted that the intermediate prefix-tree creation is done by scanning the data only once.

3.2.2.1 Example of intermediate Prefix-Tree creation.

The creation of the intermediate Prefix-Tree starts with the first entity which is the museum $M1$. A new cell is created in the root node describing the name of the country in which the museum is. The next information concerning this museum is the city where it is located. To store this information a new node will be created as a child node of the cell *Greece*. A new cell will be created in this node to store the *value city*1. The process continues until all the information about an entity are represented in the tree. When the next entity is to be inserted in the tree the insertion begins again from the root.

In figure 3.1, we give the intermediate prefix-tree of the RDF data described in section 2.

3.2.3 Final Prefix-Tree creation.

The final prefix-tree is generated from the intermediate prefix-tree by assigning the set of the possible values contained in the cells of this node to each artificial null value of a given node, if it exists. In the end, in the final prefix tree, each cell has a variable that notifies if the cell contains information coming from null cells. This information will be used on the $UNK - Finder$ and will allow us to distinguish real non keys from the undetermined ones. When the merging of the cells finishes, we proceed to the descendants of this node, and we recursively apply the processing of artificial null values and the node merge operation which is described in the following.

In figure 3.2, we give the final prefix-tree of the RDF data described in section 2.

[1] For the sake of simplicity we will use the term *value* to either refer to basic values of data properties or to URIs of object properties.

Algorithm 1 Create first version of tree

Input: RDF DataSet D , Class C
Output: $root$ of the first-version-prefix tree
$root :=$ newNode()
$P := getPropertyExpression(C, D)$
for all $C(i) \in D$ **do**
  $node := root$
  for all $PE_k \in P$ **do**
    if PE_K is inverse **then**
      $PE_k(i) := getValues(Range)$
    else
      $PE_k(i) := getValues(Domain)$
    end if
    if $PE_k(i) := \emptyset$ **then**
      if there is a $cell_1$ in $node$ with null value **then**
        $node.cell_1.URIs$.add(i)
      else
        $cell_1 := newCell()$
        $node.cell_1.value :=$"null"
        $node.cell_1.URIs$.add(i)
      end if
    else
      for all $values$ in $j \in PE_k(i)$ **do**
        if there exists $cell_1$with $value$ j **then**
          $node.cell_1.URIs$.add(i)
        else
          $cell_1 := newCell()$
          $node.cell.value := j$
          $node.cell.URIs$.add(i)
        end if
      end for
    end if
    if PE_k is not the last property **then**
      $node := cell$.child
    end if
  end for
end for

3.2.3.1 Merge Cells

In this section we describe the algorithm Merge cells, an algorithm that is used when there are nodes that contain both null and not null cells. This algorithm takes as input a node. If the node contains only one cell either it is null or not, or does not contain a null cell no changes are necessary. When a node is merged, we modify all the non null cells adding the URIs list of the null cell to them (the null cells are suppressed).

3.2.3.2 Merge Node Operation

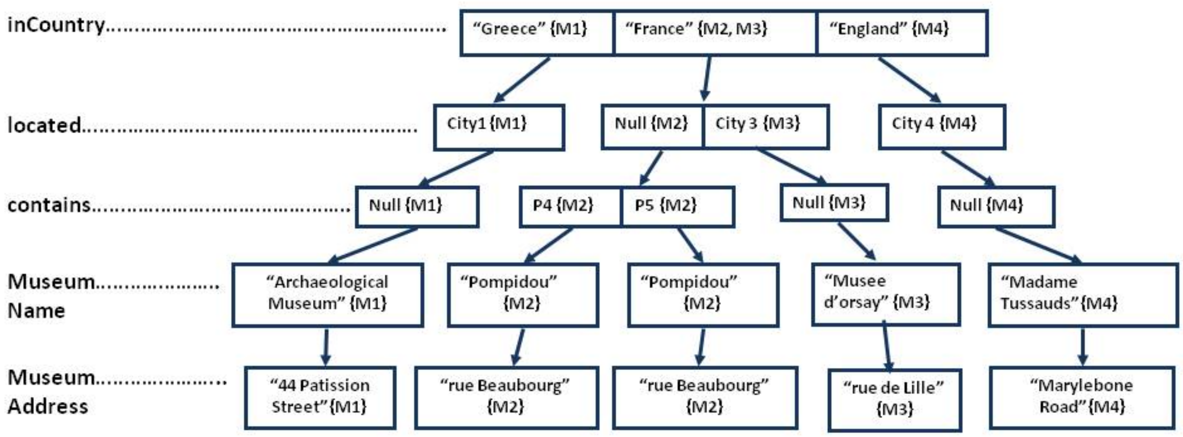

Figure 3.1: Intermediate prefix-tree for the museum class instances

Algorithm 2 Create final prefix tree

Input: $root$
$newRootNode$:= new empty $node$
$newRootNode$:= mergeCells($root$)
for all $cells$ in the $newNode$ **do**
  $cell$:=current cell
  $nodelist$:= all the children of the $cell$
  $finalChildNode$:= new node
  $finalChildNode$:= mergeNodes($nodeList$)
  $cell$.setChild = $finalChildNode$
end for
Return: $newRootNode$

This algorithm takes as input list of nodes that need to be merged and provides a merged node which contains one cell per distinct value that exists in the input list of nodes. The new URI list of

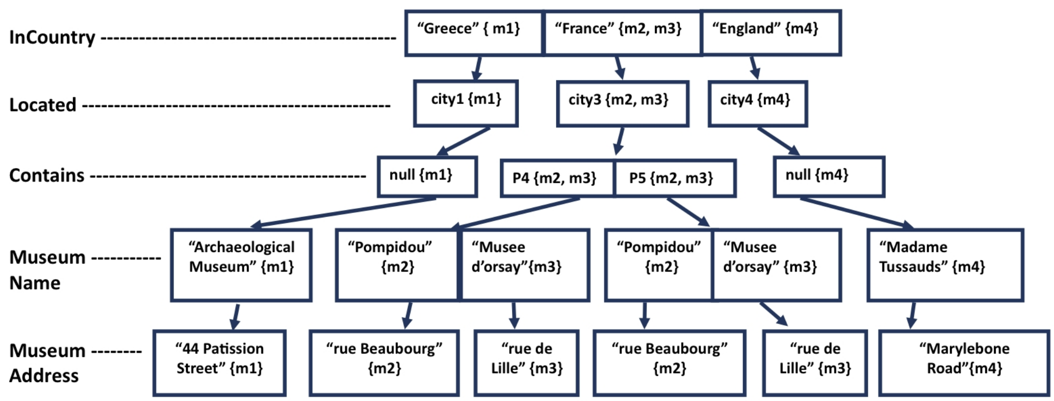

Figure 3.2: Final prefix-tree for the museum class instances

Algorithm 3 MergeCells

Input: *node*
if *node* contains only one *cell* OR *node* does not contain a *cell* with null value **then**
  $mergedNode := node$
else
  $nullCell := cell$ with null value
  for all the *cells* with value≠null **do**
    add to *cell* the URIs of *nullCell*
    add *cell* into the *mergedNode*
  end for
  $mergedNode := node$
end if
Return: $mergedNode$

---

each cell contains all the URI lists of the merged cells. In the case we have more than one nodes that need to be merged we create a new node that will contain the values from all the cells of the nodes in the list. The difference that this step introduces is that when there are cells with the same value the are compressed in only one cell with URI list all the URIs from the cells that we compressed. We use this algorithm in two different steps:

1. at the step of the creation of the final version of the tree
2. in the process of $UNK - Finder$

This merge operation is performed recursively for all the descendants of the considered nodes.The algorithm is used in two different steps, first at the step of the creation of the final version of the tree and second in the process of $UNK - Finder$.

3.2.3.2.1 Merge Node Operation Example

For example, as we can see in figure 3.1,there are two museums in France, the museums $M2$

Algorithm 4 Merge Nodes

```
Input: List of nodes to be merged
mergedNode := new empty node
for all cells in the nodeList do
  if cell.value is null then
    add cell to the nullCellList
  else
    if mergedNode contains cell.value then
      add cell.URI to the newCell.URI
    else
      newCell := new cell
      newCell.URI:= cell.URI
    end if
  end if
end for
if nullCellList is not empty then
  for all cells nullCellList do
    for all newCells in the mergedNode do
      add cell.URI to the newCell.URI
    end for
  end for
end if
if the nodeList consists of non leaf nodes then
  for each newCell do
    childrenNodeList := new list
    Add to the childrenNodeList all the children of the newCell
    newCell.setChild := mergeNodes(childrenNodeList)
  end for
end if
```

and $M3$. The museum $M3$ is located in $City3$ while there is no information about the location of $M2$. This absence of information is represented with a null cell in the node. The final node will contain only one cell which will have as a main value the $city3$. The list of the cell will be now $M2, M3$.Inside this cell will be also stored the information that this new cell contains also information coming from a null cell. This information will be used in the $UNK - Finder$ as we have already said in order to distinguish the non keys from the undetermined keys. The process of the merging will continue recursively to the children of the cells that were merged. At this time two nodes will be merged, the node with cells $P4$ and $P5$ for the museum $M2$ and the node with *null* for the museum $M3$. The final node will be a node containing two cells, $P4$ with URI list $M2, M3$ and $P5$ with URI list again $M2, M3$. Both of the cells will store the information that a null value is included in them. This process continues until there are no other merges to be executed.

3.3 Subsumption-driven Key Retrieval.

For each set of RDF sources, the method *ClassKeyRetrieval* applies a depth-first retrieval of the keys by exploiting the subsumption relation between classes declared in the ontology. *ClassKeyRetrieval* method takes an instantiated class and a possible set of already known keys as input and calculates its complete set of keys.

After creating the final prefix-tree of the considered class instances, the $UNK - Finder$ method is called for retrieving the non and undetermined keys. This algorithm is capable to distinguish the undetermined keys from the non keys. This feature of the algorithm is very important. As we already know the a undetermined key cannot be considered neither as a key nor as a non key. This means that in the it cannot affect significantly the reference reconciliation process. On the contrary, the set of non keys will be used to disincline reconciliations that are not correct.

The method for extracting both non and undetermined keys is exactly the same. For this reason, only one pass of the tree is necessary. The idea which allows the distinction of the these two sets is the existence of the null value. If a null value is contained it is a undetermined key. Otherwise it is a non key. When the method finds the sets of non and undetermined keys ,then is recursively called for the set of subclasses using the updated $knownKeysSet$.

ClassKeyRetrieval
Input: C: class; $KnownKeysSet$:=set of known keys
Output: $CKeys$: the complete set of keys of the class C.
if $class$ has declared properties **then**
  $tripleList$.add(all triples of C)
  if $tripleList$ is not empty **then**
    $rootNode$:= Create-intermediate-prefix-tree(tripleList, C)
    $newRootNode$:=Create-final-prefix-tree($rootNode$)
    $propNo := 0$
    UNK_{CS} :=UNK-Finder($newRootNode$,$propNo$, $KnownKeys$)
    $UK_{CS} := UNK_{CS}$.UKeySet
    $NK_{CS} := UNK_{CS}$.NKeySet
    $keys := ExtractKeysFromUNKeySet(UNkeySet, C)$
    $CKeys := refine(KnownKeysSet.add(keys))$
  end if
end if
for all subClass C_i of C **do**
  ClassKeyRetrieval(C_i, $CKeys$)
end for
return $CKeys$

3.3.1 UNK-Finder: UK_{CS} and NK_{CS} Finder.

The UNK_{CS} is the set of non and undetermined keys. The process of the algorithm begins from the root of the prefix tree and makes a depth-first traversal of it. The input of the algorithm is the current root of the tree, its attribute number and the known keys. This method searches the longest path p from the root to a node having a URI list containing more than one URI. p represents the maximal set of properties expressing either a non key or undetermined key.

To ensure the scalability of the key discovery, KD2R performs three kinds of pruning:

1. the subsumption relation, which is used between classes is exploited to prune the key discovery thanks to the set of inherited keys,

2. the anti-monotonic characteristic of the undetermined keys and non keys which is used to avoid computing the redundant undetermined and non keys, i.e. if {ABC} is a non key (resp. an undetermined key) then all the subsets of {ABC} are also non keys (resp. an undetermined key) and

3. the monotonic characteristic of keys which is used to avoid exploring the descendants of a node representing only one instance.

This algorithm performs a traversal of the tree in order to find the maximal undetermined and non keys. The UNK-Finder takes as input the root of the tree, the attribute number and the already known keys.The variable $curUNKey$ is global and represents the candidate undetermined or non key that is tested each time. This algorithm is designed in such a way to be able to avoid the production of redundant non and undetermined keys. since the goal is to find the maximal ones.

When the UNK-Finder visits a node, it adds the $propNo$ to the $curUNKey$ and then proceeds to the contents of the node. AS we can see in the algorithm after this step the $propNo$ is removed from the $curUNKey$ the children of the node are merged and then the UNK-Finder is executed for the new mergedTree.

In case the UNK-Finder proceeds to a leaf if the $URIList$ is bigger than 1 this means that the are more than two instances with the same values and so the $curUNKey$ will be added to either the NK_{CS} or to the UK_{CS}. In order to be able to separate the non keys from the undetermined ones we have to test if one of the cells that participate in the $curUNKey$ has come from a merge with a null value. In case that this happens this means that the $curUNKey$ will be added in the set UK_{CS}. Otherwise it will be part of the NK_{CS}. The algorithm continues by removing the $propNo$ from the $curUNKey$. If the current root has more than one cell and at least one of these cells has $URIList$ bigger than 1 the $curUNKey$ will be added to either NK_{CS} or UK_{CS} using exactly the same way to decide in which it will be inserted.

3.3.1.1 Example of UNK-Finder.

We illustrate the UNK-Finder algorithm on the final prefix-tree shown in figure 3.2. The method begins with the first node and more specifically with the cell containing the value " *Greece*". The property number of the cell, 0, is added on to $curUNKey$. Since the $URIList$ of this cell has size one ($M1$) -thanks to the pruning step- the algorithm will not examine its children. The meaning of the pruning step is that since a cell has size of the $URIList$ is 1 this means it describes only one instance. Therefore, no undetermined key or non key can be found. This is the reason why there is no interest in testing the children of this node. Now the property number is removed and the $curUNKey$ is empty.

The algorithm moves to the next cell of the root node, containing the value " *France*". The property number 0 is added in the $curUNKey$. This cell contains a $URIList$ with two elements in it, $URIList = M2, M3$. Recursively, we go to child node with cell "*city3*".The property number of the cell is added in the $curUNKey$and now the $curUNKey$ is $(0, 1)$. We call the UNK-Finder for the child node of the "*city3*". Now the root node is the node with paintings $P4$ and $P5$ and the property number of the node is added to the $curUNKey$ $(0, 1, 2)$.

The process continues with the child node of cell $P4$. In the $curUNKey$, property number 3 is added. Since the cell "Pompidou" has $URIList$ of size one the UNK-Finder will not continue with the child node of "*Pompidou*" thanks to the pruning technique.3 will be removed from the $curUNKey$. The method will continue with the second cell of the root node which is "Musee d'Orsay". Property number 3 will be added again in $curUNKey$ which will be now $0, 1, 2, 3$. Like "*Pompidou*", and since "*Musee d'Orsay*" has $URIList$ size one, the UNK-Finder will not be called for its child node.3 will be removed again from the $curUNKey$.

The UNK-Finder has been called for each cell of the node. The property number of the node is removed from the $curUNKey$ which now is $(0, 1, 2)$. In this step the child nodes of this node are merged and the UNK-Finder is applied to the mergedTree.

The UNK-Finder is executed for the new merged node which consists of two cells, "*rue Beaubourg*" with $URIList = M2$ and "*rue de Lille*" $URIList = M3$. The property number of the merged

UNK-Finder

Input: *root*: node of the prefix tree; $propNo$: attribute Number; $knownKeysSet$: given keys.
Output: NK_{CS},UK_{CS}: the set of discovered non and undetermined keys.

add $propNo$ to the $curUNKey$
if *root* is a leaf **then**
  for all *cells* in the *root* **do**
    if $cell.URIList > 1$ **then**
      if one of the cells that participate in the $curUNKey$ comes from a merge with null value **then**
        add $curUNKey$ to the NK_{CS}
      else
        add $curUNKey$ to the UK_{CS}
      end if
      break
    end if
  end for
  remove $propNo$ from $curUNKey$
  if *root* has more that one cell AND at least one of the cells has $URIList > 1$ **then**
    if one of the cells that participate in the $curUNKey$ comes from a merge with null value **then**
      add $curUNKey$ to the NK_{CS}
    else
      add $curUNKey$ to the UK_{CS}
    end if
  end if
else
  if there is only one URI **then**
    return
  end if
  for all *cells* in the *root* **do**
    if $curUNKey$ is not contained in $knownKeysSet$ **then**
      UNK-Finder($cell.getChild$,$propNo$+1)
    end if
  end for
  remove $propNo$ from $curUNKey$
  if $curUNKey$ is already contained in the UNK_{CS} **then**
    return
  end if
  $childNodeList :=$ all the children of the cells in the *root* node
  $mergedTree := mergeNodes(childNodeList)$
  if $curUNKey$ is not contained in $knownKeysSet$ Set **then**
    UNK-Finder($mergedTree$,$propNo$+1)
  end if
end if
return UNK_{CS}

node is added in the $curUNKey$ which is $(0,1,2,4)$. Since we are in the leaf and none of the cells has $URIList$ bigger than 1 the property number 4 is removed from the $curUNKey$.

Now the $curUNKey$ is $(0,1,2)$. Since the root has more than one cells the $curUNKey$ will be added either to the NK_{CS} or to the UK_{CS}. To decide in which set we should add the $curUNKey$ we have to check if at least one of the cells that participate in the $curUNKey$ comes from a merge with a null value. The $curUNKey$ is finally added in the UK_{CS}. So $\{inCountry, located, contains\}$ is an undetermined key.

Since the all the cells of the current node -$P4$ $P5$- have been tested, the property number 2 is removed from the $curUNKey$. Now the $curUNKey$ is $0,1$. Recursively the algorithm will detect one non key which is $\{inCountry\}$ - is a non key since there are two museums in France-.

3.3.2 Extraction of K_{CS} from UNK_{CS}.

In order to compute the set of minimal k_{CS} we need a set that contains both the NK_{CS} and the UK_{CS}. This set is declared in the UNK-Finder as UNK_{CS}. To proceed to the computation of the K_{CS}, for each undetermined or non key that appears in the UNK_{CS} we calculate the complement set. Then we apply the Cartesian product on the obtained complement sets. Finally, we remove the non-minimal k_{CS} from the obtained multi-set of k_{CS}.

Extraction of K_{CS} from UNK_{CS}
Input: UNK_{CS}: container of non and undetermined keys
Output: K_{CS}: set of the keys
K_{CS}:=0
for all undetermined or non keys that in the UNK_{CS} **do**
  complementSet:= complement of the undetermined or non key
  if K_{CS}:=0 **then**
    K_{CS}:= *complementSet*
  else
    newSet:=0
    for all pk_{CSi} in the *complementSet* **do**
      for all k_{CS} in the K_{CS} **do**
        insert (pk_{CS} union k_{CS}) into *newSet*
      end for
    end for
    simplify *newSet*
    K_{CS}:= *newSet*
  end if
end for

3.3.2.1 Example of the extraction of K_{CS} from UNK_{CS}.

In the museum example we have two undetermined or non keys which are $\{contains, located, inCountry\}$ and $\{inCountry\}$ which is already contained in the first one. Since the only of the two undetermined or non keys is a subset of the other we will use only the maximal as we have already mention in this report. The complement set of this undetermined or non key is $\{MuseumAddress\}$,$\{MuseumName\}$. The process finishes by adding the two keys to the K_{CS}. The keys in the K_{CS} are $MuseumAddress$ and $MuseumName$.

3.4 Complexity.

The calculations of the complexity of UNK-Finder and extraction of K_{CS} from UNK_{CS} is based on [10]. In general we know that retrieving minimal composite keys is a NP-complete problem [5]. To compute the complexity we make the same assumptions as in [10]:

1. Each attribute appears in a data set with frequency that follows the Zipfian distribution [2] with parameter q, so that the frequency of the ith most frequent value is proportional to i^{-q}.

2. Our data do not have correlations even if in the real world this may happen very often.These correlations would have improved the complexity if the had been taken into account.

Under these assumptions the time complexity of UNK-Finder and extraction of K_{CS} from UNK_{CS} for one file and only one class is:

$$C_ent = O(s * d * T^{1+\frac{(1+q)}{\log_d C}} + s^2)$$

where s is the number of mutually non keys and undetermined keys, d is the number of attributes, C is the average cardinality (number of distinct values) of the attributes, and T is the number of entities.The term s^2 expresses the cost of computing the K_{CS} from the UNK_{CS} and uses the fact that the number of keys is $O(s)$. The complexity of KD2R is:
$C = C_ent * K * F + K * z^2$

where K is the number of entities in a file, F the number of files we have and z the maximal number of keys found in one file. Even if all the assumptions we make do not always hold the performance of KD2R is clearly superior than the exponential time and the polynomial space requirements of the brute-force approach.

[2]Zipf's law states that given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table. Thus the most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

CHAPTER 4

First experiments

We have implemented and tested our method on two datasets that have been used in the Ontology Alignment Evaluation Initiative (OAEI, http://oaei.ontologymatching.org/2010/). UNA is declared for each RDF file of the two datasets. Since the two ontologies have been enriched by expert keys, we have compared our results to the set of these existing keys.

4.1 Datasets description.

The first dataset $D1$ describes 1729 instances (classes *Restaurant* and *Address*) as it is illustrated in 4.2. In the provided Ontology, Restaurants are described using the following properties: *name*, *phoneNumber*, *hasCategory*, *hasAddress*. Addresses are described using: *street*, *city*, *Inverse*(*hasAddress*) (4.1). The first RDF file f1 describes 113 *Address* instances and 113 *Restaurant* instances. The second RDF file f2 describes 641 *Restaurant* instances and 752 *Address* instances.

The second dataset D2 consists of 3200 instances of *Person* and *Address* describing people as we can see in 4.2. In the ontology, a person is described by the following properties: *givenName*, *state*, *surname*, *dateOfBirth*, *socSecurityId*, *phoneNumber*, *age* and finally *hasAddress*. An Address is described by the properties: *street*, *houseNumber*, *postcode*, *isInSuburb* and finally *inverse*(*hasAddress*)(4.1). The first and the second RDF files contain each of them 500 instances of the class *Person* and 500 instances of *Address*. The third file contains 600 *Person* instances and 600 *Address* instances.

Table 4.1: Experiments table 1

Datasets	Classes	Property Set
Restaurants (2 files)	Restaurant	name, phoneNumber, hasCategory, hasAddress
	Address	street, city, Inverse(hasAddress)
Person (3 files)	Person	givenName, state, surname, dataOfBirth, socSecurityId, phoneNumber, age, hasAddress
	Address	street, houseNumber, postcode, isInsuburb

Table 4.2: Experiments table 2

Datasets	RDF Files	instances
Restaurants Dataset	Restaurant1.rdf	339
	Restaurant2.rdf	1390
Person Dataset	Person11.rdf	1000
	Person12.rdf	1000
	Person21.rdf	1200

4.2 Obtained results

To examine the results of our method we compared the KD2R keys with the keys given by an expert. 10% of found keys are equal to the expert keys and 10% are bigger (i.e., contain more properties). The first case is the best we can come up with since our results agree with the expert ones. The second case arises when an expert makes a mistake and declares as keys properties that are not in fact real keys. This means that we detect erroneous keys given by an expert. For instance, the expert has declared that phoneNumber is a key. We are sure that the expert has made a mistake since in our data we can find two different restaurants with the same phone number (managed by the same organization). These two cases (20% of our found keys) represent the definite minimal keys that we extract using the given datasets. Another 20% of KD2R keys are keys that are smaller compared to the expert keys. It is possible to face this case when the given data are not sufficient to find more specific keys. Finally the 60% of the found keys are keys that are not declared by the expert. For example we find that $Inverse(hasAddress)$ can be a key for the address, a property that the expert did not take into account and seems to be relevant (a museum has only one address).

Thus, KD2R may find keys that are not specific enough (the more the data are numerous the more the discovered keys are accurate). However, this method can also find keys that are equal to the expert ones or keys which are missed by the expert.

CHAPTER 5

Related works

The reference reconciliation problem consists on whether different references refer to the same real world entity. Many approaches (see [4] or [12] for a survey) try to reconcile data. Recent global approaches exploit the existing dependencies between reference reconciliation decisions [9, 3, 1]. In such approaches, the reconciliation of one reference pair may entail the reconciliation of another reference pair. A knowledge based approach is an approach in which an expert is required to declare knowledge that will be used by the reference reconciliation system [7, 3]. Some approaches such as [7] use reconciliation rules that are given by an expert, while other approaches such as [9] use the (inverse) functional properties (or the keys) that are declared in the ontology. Nevertheless, when the ontology represents many concepts and when data are numerous, such keys are not easy to model for the ontology expert.
The problems of key discovery in OWL ontologies and key discovery or Functional Dependency discovery in relational databases are very similar. In the relational context, key discovery is a sub-problem of extracting functional dependencies (FDs) from the data. [11] proposes a way of retrieving probabilistic FDs from a set of data sources. Two strategies have been proposed: the first one merges the data before discovering FDs while the second one merges the FDs obtained from each data source. These probabilistic FDs are used to identify data sources that do not conform to the FDs that are found. Also, since the sources that are united are not always described using the same schema, the new mediated schema that is created can be normalized using these probabilistic FDs. This paper focuses on the problem of finding probabilistic FDs with only a single attribute in each side.In order to find the FDs, TANE [6] partitions the tuples into groups based on their attribute values. The goal is to find approximate functional dependencies: functional dependencies that almost hold. In the context of Open Linked Data, [8] have proposed a supervised approach to learn functional dependencies on a set of reconciled data.

There are a lot of works that deal with the discovery of FDs in relational databases, however only a few of them focus on the specific problem of retrieving keys. The Gordian method [10] allows discovering composite keys in relational databases. In order to avoid to checking all the possible combinations of candidate keys, the method proposes the discovery of the non-keys in a dataset and then using them to find the keys. In this method a prefix tree is built and explored (using a merge step) in order to find the maximal non keys. To optimize the tree exploration, they exploit the anti-monotone property of a non key. Nevertheless, it is assumed that the data are completely described (without null values). Furthermore, multivalued attributes are not taken into account.

The approach we propose allows dealing with incomplete data that are described using possibly multivalued properties. Furthermore, since the approach is proposed for RDF resources conform with a OWL2 ontology, KD2R exploits the subsumption relation that may exist between classes. KD2R aims to find keys that are exact wrt a set of data set and a given class and do not aims to find probabilistic keys.

CHAPTER 6

Conclusions and Future work

In this paper, we have described the method KD2R which aims to discover keys in RDF data in order to use them in a reconciliation method. These data conform to the same ontology and are described in RDF files for which the UNA is fulfilled. The approach can also be used to help an expert to define or enrich a set of keys. KD2R takes into account the properties that the RDF data sets may have : incompleteness and multi-valuation. Since the data may be numerous, the method discovers maximal non keys and undetermined keys that are used to compute keys and merge them if keys are discovered using different datasets. Furthermore, the approach exploits key inheritance due to subsumption relations between classes to prune the key search for a given class. The first experiments have been conducted on two datasets exploited in the OAEI evaluation initiative. We have compared the retrieved keys with keys given by an expert. Some of the found keys are less specific than the expert ones but errors of the expert can also be detected.

We plan to test our approach on bigger datasets. It will be then interesting to compare the reconciliation results, using LN2R, when KD2R key constraints are considered, with the results that are obtained without using keys. We also plan to test our approach on more heterogeneous data. We aim at extending our method in order to be able to work when the UNA is not fulfilled. Indeed the syntactic variations can affect the key discovery process. If similarity measures or lexical resources are used, the key discovery method has to take into account the similarity scores.

Acknowledgements

First of all I would like to thank my supervisors, Nathalie Pernelle and Fatiha Saïs for giving me the opportunity to work with them and to be a part of the IASI/Leo team. The last 6 months of my life I have been more productive than ever in my whole life, and their behaviour played an important role in that. Without their support and guidance my master thesis would not be the same. They also motivated me to continue my studies and become a phD student.
I would also like to thank them for trusting me and for treating me in such a great way. They will always be the best professors I have ever worked with.
To continue I would like to thank the IASI/Leo team for supporting me and for making me feel as a part of the team from the first time. Being a part of a great team gave me inspiration and courage to overcome all the difficulties that I may have been through.
I would like to thank also the University of Crete which gave me the opportunity to be a part of it through the Greek-French Postgraduate Program. More specifically I would like to thank all my professors in the University of Crete for the support and for the great things that they have learned me the last two years.
I would like also to thank my professor Kostas Margaritis, in the University of Macedonia, for being such a great professor and for his support. I would always remember the great advises he gave to me, and the fact that I can always count on his help.
Moreover I would like to thank many people from Greece and France for being there for me and for supporting me through my master. More specifically I would like to thank Stamatis, Georgia, Mary, Vicky and Sofia for helping me and for being by my sight.
Special thanks to Nelly Vouzoukidou, the person that helped me more than anyone else, and who believed in me more than I did.
Last but not least, I would like to thank my family for being always by my sight and especially my mother who is the person that is always there to listen to all my crazy thoughts. I dedicate this work in my family and more specifically in my mother Zina, my father Kostas, my sister Eleni and my brother Hraklis.

Η εργασία αυτή είναι αφιερωμένη στην οικογένεια μου, τον πατέρα μου Κώστα, την αδερφή μου Ελένη, τον αδερφό μου Ηρακλή και ειδικά στην μητέρα μου Ζήνα, μιας και χωρίς αυτούς τίποτα δεν θα ήταν το ίδιο!

Bibliography

[1] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1, March 2007.

[2] Nathalie Pernelle Danai Symeonidou and Fatiha SaÃ¯s. Kd2r : a key discovery method for semantic reference reconcilation. SWWS '11, Springer LNCS, 20011.

[3] Xin Dong, Alon Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *Proceedings of the 2005 ACM SIGMOD*, SIGMOD '05, pages 85–96, NY, USA, 2005.

[4] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection : A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19 :1–16, 2007.

[5] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28 :140–174, June 2003.

[6] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane : An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2) :100–111, 1999.

[7] Wai Lup Low, Mong Li Lee, and Tok Wang Ling. A knowledge-based approach for duplicate elimination in data cleaning. *Information Systemes*, 26 :585–606, December 2001.

[8] Andriy Nikolov and Enrico Motta. Data linking : Capturing and utilising implicit schema-level relations. In *Proceedings of Linked Data on the Web workshop collocated with WWW'2010*, 2010.

[9] Fatiha Saïs, Nathalie Pernelle, and Marie-Christine Rousset. Combining a logical and a numerical method for data reconciliation. *Journal on Data Semantics, vol 12*, pages 66–94, 2009.

[10] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. Gordian : efficient and scalable discovery of composite keys. In *Proceedings of the 32nd International conference VLDB*, VLDB '06, pages 691–702. VLDB Endowment, 2006.

[11] Daisy Zhe Wang, Xin Luna Dong, Anish Das Sarma, Michael J. Franklin, and Alon Y. Halevy. Functional dependency generation and applications in pay-as-you-go data integration systems. In *12th International Workshop on the Web and Databases*, 2009.

[12] William E. Winkler. Overview of record linkage and current research directions. Technical report, Bureau of the Census, 2006.