Automatic key discovery for Data Linking

Danai Symeonidou

UNIVERSITÉ PARIS SUD

ÉCOLE DOCTORALE INFORMATIQUE DE PARIS SUD

Laboratoire de Recherche en Informatique (LRI)

***DISCIPLINE** INFORMATIQUE*

THÈSE DE DOCTORAT

presentée en vue d'obtention du titre de docteur

par **Danai SYMEONIDOU**

Automatic key discovery for Data Linking

Directeur de thèse :	Nathalie Pernelle	Université Paris Sud
Co-directeur de thèse :	Fatiha Saïs	Université Paris Sud

Composition du jury :

Rapporteurs :	Marie-Christine Rousset	Université de Grenoble
	Aldo Gangemi	Université Paris 13
Examinateurs :	Olivier Curé	Université Marne-la-Vallée
	Alain Denise	Université Paris Sud

To my mom Zina and my dad Kostas for
their unconditional love and support...

Abstract

In the recent years, the Web of Data has increased significantly, 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 data. Among the different kinds of semantic links that can be established, *identity links* express that different resources refer to the same real world entity. By comparing the number of resources published on the Web with the number of identity links, one can observe that the goal of building a Web of data is still not accomplished. Several data linking approaches infer identity links using keys. Nevertheless, in most datasets published on the Web, the keys are not available and it can be difficult, even for an expert, to declare them.

The aim of this thesis is to study the problem of automatic key discovery in RDF data and to propose new efficient approaches to tackle this problem. Data published on the Web are usually created automatically, thus may contain erroneous information, duplicates or may be incomplete. Therefore, we focus on developing key discovery approaches that can handle datasets with numerous, incomplete or erroneous information. Our objective is to discover as many keys as possible, even ones that are valid in subparts of the data.

We first introduce KD2R, an approach that allows the automatic discovery of composite keys in RDF datasets that may conform to different schemas. KD2R is able to treat datasets that may be incomplete and for which the Unique Name Assumption is fulfilled. To deal with the incompleteness of data, KD2R proposes two heuristics that offer different interpretations for the absence of data. KD2R uses pruning techniques to reduce the search space. However, this approach is overwhelmed by the huge amount of data found on the Web. Thus, we present our second approach, SAKey, which is able to scale in very large datasets by using effective filtering and pruning techniques. Moreover, SAKey is capable of discovering keys in datasets where erroneous data or duplicates may exist. More precisely, the notion of *almost keys* is proposed to describe sets of properties that are not keys due to few exceptions.

Keywords : Semantic Web, RDF, Ontologies, OWL, Linked Data, Data linking, Key Discovery, Scalability, Erroneous data, Incomplete data

Résumé

Dans les dernières années, le Web de données a connu une croissance fulgurante arrivant à un grand nombre des triples RDF. Un des objectifs les plus importants des applications RDF est l'intégration de données décrites dans les différents jeux de données RDF et la création des liens sémantiques entre eux. Ces liens expriment des correspondances sémantiques entre les entités d'ontologies ou entre les données. Parmi les différents types de liens sémantiques qui peuvent être établis, les *liens d'identité* expriment le fait que différentes ressources réfèrent au même objet du monde réel. Le nombre de liens d'identité déclaré reste souvent faible si on le compare au volume des données disponibles. Plusieurs approches de liage de données déduisent des liens d'identité en utilisant des clés. Une clé représente un ensemble de propriétés qui identifie de façon unique chaque ressource décrite par les données. Néanmoins, dans la plupart des jeux de données publiés sur le Web, les clés ne sont pas disponibles et leur déclaration peut être difficile, même pour un expert.

L'objectif de cette thèse est d'étudier le problème de la découverte automatique de clés dans des sources de données RDF et de proposer de nouvelles approches efficaces pour résoudre ce problème. Les données publiées sur le Web sont général volumineuses, incomplètes, et peuvent contenir des informations erronées ou des doublons. Aussi, nous nous sommes focalisés sur la définition d'approches capables de découvrir des clés dans de tels jeux de données. Par conséquent, nous nous focalisons sur le développement d'approches de découverte de clés capables de gérer des jeux de données contenant des informations nombreuses, incomplètes ou erronées. Notre objectif est de découvrir autant de clés que possible, même celles qui sont valides uniquement dans des sous-ensembles de données.

Nous introduisons tout d'abord KD2R, une approche qui permet la découverte automatique de clés composites dans des jeux de données RDF pour lesquels l'hypothèse du nom Unique est respectée. Ces données peuvent être conformées à des ontologies différentes. Pour faire face a‘ l'incomplétude des données, KD2R propose deux heuristiques qui permettent de faire des hypothèses différentes sur les informations éventuellement absentes. Cependant, cette approche est difficilement applicable pour des sources de données de grande taille. Aussi, nous avons développé une seconde approche, SAKey, qui exploite différentes techniques de filtrage et d'élagage. De plus, SAKey permet à l'utilisateur de découvrir des clés dans des jeux de données qui contiennent des données erronées ou des doublons. Plus précisément, SAKey découvre des clés, appelées "almost keys", pour lesquelles un nombre d'exceptions est toléré.

Mots clés : Web Sémantique, RDF, Ontologies, OWL, Linked Data, Liage de Données, Découverte de Clés, Passage à l'échelle, Données Incomplètes, Données Erronées

Acknowledgements

I would like to express my sincerest gratitude to my supervisors Nathalie Pernelle and Fatiha Saïs for their guidance, support and patience during all these years. Since the beginning, both Fatiha and Nathalie trusted me and treated me as equal. They provided me the right balance between supervision and freedom, accepting me the way I am. It is mainly thanks to them that I want to continue my career in scientific research. In particular, Fatiha has provided me with positive energy and encouraged me in these first steps in research. Nathalie has been always there, even devoting her personal time during weekends, to help me. She did not only have the role of a mere supervisor but she made me feel part of her family. We shared a lot of everyday moments, from helping me move to a new apartment to eating together lots of chocolates after Sunday work sessions!

I am grateful to Marie-Christine Rousset and Aldo Gangemi for having thoroughly read my thesis and providing me with constructive remarks. I would like to thank also my thesis committee, Alain Denise and Olivier Curé for honoring me with their presence in my defense.

Being a member of the Qualinca project has been a great experience for me. I would like to thank all the members of the project for the great work we did together. This project gave me the chance to meet great people, exchange ideas and discuss; a process that provided me with enthusiasm and motivation for this PhD.

I am grateful for having met and collaborated with Nicole Bidoit. She has trusted me with teaching even when I was not trusting myself enough and has always been there willing to answer my every single question. As a professor, I consider her a role model for the way she organizes her courses and the responsibility she shows towards her students. Nicole has made my life easier whenever I had to face administrative issues and has repeatedly gone out of her way to help me.

To continue I would like to thank the LaHDAK team for supporting me and for making me feel as a part of the team from the first moment. Being a part of a great team gave me inspiration and courage to overcome all the difficulties that I may have been through. I want to thank especially Chantal Reynaud, the Head of the LaHDAK team for her support, trust and help in anything I have asked for. Stamatis, Jesús, Juan, Benjamin, Damian, Soudip,

Sejla, Raphael, Alexandra, Francesca, Elham and some former colleagues but still great friends, Gianluca, Andre, Asterios, Fayçal, Despoina, Zoi, Tushar and Andrés created a lovely atmosphere in the lab and offered me unforgettable moments of fun.

I am grateful for having had the opportunity to meet and share my office with Laura Papaleo. Laura has been a source of positive energy in the lab. Her research experience and wide knowledge in multiple domains have rendered all our discussions very interesting and productive. I have especially appreciated her selfless willingness to help me in any occasion. Her valuable feedback has highly improved the quality of my thesis.

Being surrounded by very good friends made all my life in Paris even more beautiful. Thanks Nelly, Dimitris, Fabien, Stamatis, Marios, Foivos and Giorgos for all the great moments that we shared. Lia, Hara, Maria, Evi and Marina thanks for being in my life even if we were physically so far away.

I would like to thank Mary Koutraki for being a true friend; During these years we have shared and discussed about nearly everything, research-oriented and beyond. I have always admired her open-mindedness, determination and clear way of thinking and I hope we can one day manage to work in the same team.

I would also like to thank Katerina Tzompanaki, my teammate, as I used to call her, for being such a good friend. I did my best to give her back all the constant support, positive energy and help she has offered me during these years. The fact that Katerina never gives up has been an inspiration for me.

Sharing the office with Ioanna Giannopoulou has been one of the greatest periods in this lab. Ioanna had the "chance" to share an office with me during the last and the most difficult year of this PhD so I want to thank her for her patience! Having Ioanna next to me made everything look easier.

Katerina, Mary and Ioanna have been my best reviewers for anything I have written, presented and thought during these years.

I would like to thank Vincent Armant, my old deskmate, my best friend, my number-one supporter. Sharing the office with Vincent has been an amazing experience for all the incredibly fun moments we had together. Our endless discussions about research topics but also about everyday life have been priceless. Vincent has helped me to take distance from my topic and regain motivation in difficult times. Our discussions have led to a very fruitful collaboration. He has been my translator and personal French tutor. For his support, both moral and practical, and for all the things I keep learning from him on every level, I will be forever grateful.

Finally, I want to thank my family for their love and support along these years. With-

out them, none of these would have been possible. Eleni, my sister and best friend, has been the person that has listened to my craziest thoughts and with who I have made endless discussions about every single thing. My brother, Iraklis, has always been there when big decisions for my life had to be made. Last but not least, my role models, the most important people in my life; my parents Zina and Kostas. Without their support, love and encouragement I would not be the same person. I am dedicating this thesis to them. My father, my personal trainer as I often call him, has taught me to love science, to always try for the best and to never give up. My mother has always been there for me and supported me in every possible way. She taught me to see the bright side of things and she is the person that brings sun in my life. Finally, I thank them both for teaching me that the "key" in life is to make ourselves better than we were yesterday and not better than others.

Résumé de la thèse en français

Introduction

Au cours des dernières années, le nombre de données RDF disponibles sur le Linked Open Data cloud (LOD) a cru très rapidement. En Juillet 2014, le nombre de triplets dans le LOD a dépassé les 61 milliards [1]. Dans cet espace de données, le liage d'informations provenant de différentes sources de données est un point crucial. Établir des liens sémantiques entre des données décrites dans différents sources permet de transformer des données locales en un espace global des données. En effet, une fois que les données sont liées, elles peuvent être consommées pour des applications émergentes, tels que les navigateurs de données liées, des moteurs de recherche, des crawlers ou des applications spécifiques à un domaine particulier.

Les liens sémantiques peuvent être indépendants du domaine(en représentant, par exemple, que deux éléments réfèrent au même objet du monde réel), ou dépendants du domaine (en dénotant, par exemple, que la ville de "Paris" est située près de la ville de "Versailles").

En comparaison avec le grand nombre de triplets disponibles sur le Web, le nombre de liens existants entre ces triplets est très faible. En effet, il existe environ 643 millions de liens pour 61 milliards de triplets. La découverte et la publication de nouveaux liens dans le LOD est, de nos jours, un sujet de recherche très actif.

Les liens sémantiques peuvent être déterminés manuellement par un expert. Cependant, compte tenu de la grande quantité de données disponibles sur le Web, la création manuelle de liens sémantiques n'est pas envisageable. Parmi les différents types de liens sémantiques qui peuvent être établis, les liens d'identité, expriment le fait que différentes ressources se réfèrent à la même entité du monde réel. Par exemple, un lien d'identité entre deux instances de personnes indique que ces deux instances se réfèrent à la même personne. La plupart des liens entre les différents jeux de données du LOD aujourd'hui sont essentiellement des liens d'identité.

Dans la littérature, il existe un grand nombre des approches qui visent à détecter les liens

[1] http://stats.lod2.eu

d'identité entre des données. De nombreuses approches utilisent des règles pour spécifier les conditions que les deux données doivent remplir pour être liés. Ces règles peuvent être très spécifiques et exprimées en utilisant différentes mesures de similarité, des pondérations et des seuils.

D'autres approches sont fondées sur des règles logiques, comme la fonctionnalité (inverse) de propriétés. Quelle que sait le type d'approche, les règles s'appuient sur des propriétés discriminatives ou clés. Une clé représente un ensemble de propriétés qui identifie chaque ressource de manière unique. En d'autres termes, si deux ressources partagent des valeurs pour toutes les propriétés d'une clé, alors ils se réfèrent à la même entité du monde réel.

Néanmoins, lorsque les propriétés et les classes sont nombreuses, les clés ne peuvent pas être facilement spécifiées par un expert humain. Il est difficile de supposer qu'un expert est toujours disponible pour spécifier des clés pour chaque jeu de données et chaque domaine d'application. En outre, cette tâche devient encore plus difficile quand un ensemble complet de clés composites est nécessaire. En effet, lorsque les données sont incomplètes, plus les clés sont nombreuses et impliquent des propriétés différentes, plus les décisions qui peuvent être prises dans le processus de liage de données sont nombreuses. Le problème de découvrir l'ensemble des clés automatiquement à partir de les données est #P-hard. Par conséquent, des approches en mesure de découvrir des clés efficacement sont essentielles.

De plus, étant donné que les données disponibles sur le Web sont construites de façon autonome et se conforment aux différents domaines d'applications, les jeux de données sont, par construction, hétérogènes et peuvent contenir des erreurs et des doublons. Dans ce cas, le problème de découverte de clés devient beaucoup plus complexe. En outre, dans le contexte du Web sémantique, les données RDF peuvent être incomplètes, et l'affirmation du *Closed World Assumption* (précisant que ce qui n'est pas indiqué comme étant vrai est faux) peut ne pas être significative. Par conséquent, il est nécessaire de concevoir de nouvelles stratégies qui découvrent des clés dans de grands jeux de données RDF qui peuvent être corrompu et incomplets.

Dans cette thèse, nous étudions les aspects théoriques et pratiques de la découverte automatique de clés dans les données RDF. Plus précisément, nous avons pour objectif de développer des approches de découverte de clés qui sont en mesure de traiter des jeux de données où les données peuvent être nombreuses, incomplètes et erronées. Le but est de découvrir autant de clés que possible, même celles qui ne s'appliquent pas à l'ensemble du jeu de données.

État de l'art & motivation

Les liens d'identité entre les ressources de différents jeux de données ajoutent une valeur réelle pour les données liées existant sur le Web. En raison de l'énorme quantité de données disponibles et à son hétérogénéité, il est difficile de déclarer manuellement des liens d'identité. Par conséquent, il existe de nombreuses approches qui mettent l'accent sur la découverte automatique de liens d'identité.

Certaines approches sont numériques et utilisent des mesures de similarité complexes, fonctions d'agrégation et seuils pour construire les règles. Ces approches sont souvent adaptées à quelques jeux de données. Dans certains cas, ce type de règles est automatiquement découvert [IB12, NL12, NdM12, SAS11]. D'autres approches sont basées sur des règles logiques [SPR09, HCQ11] qui sont générés automatiquement en utilisant la sémantique des clés ou des propriétés (inverse) fonctionnelles. Ce type de connaissance peut également être utilisé par un expert pour construire des fonctions de similarité plus complexes qui prennent plus en compte les propriétés qui sont impliquées dans les clés [VBGK09, NA11]. Pour des raisons d'évolutivité, les clés peuvent aussi être impliquées dans les méthodes permettant d'identifier des blocs qui probablement réfèrent au même objet du monde réel [SH11].

L'avantage d'utiliser des clés au lieu des fonctions de similarité complexes est que les clés peuvent être valides pour un domaine et pas seulement pour une paire des jeux de données spécifique. Comme il peut être difficile, même pour un expert de définir l'ensemble des clés et qu'une telle connaissance n' est généralement pas déclarée dans une ontologie, des approches qui découvrent automatiquement des clés à partir des données sont nécessaires.

Le problème de la découverte clé a déjà été étudié dans le cadre des bases de données relationnelles. Comme indiqué précédemment, la découverte de clés minimales composites (i.e., des clés minimales qui sont composées de plus d'un attribut) dans un jeu de données est #P-hard [GKM$^+$03]. En effet, pour vérifier si un ensemble de propriétés est une clé, une approche naïve serait de numériser tous les tuples d'une table pour vérifier se il n'y a pas au moins deux tuples partageant les même valeurs pour un sous ensemble d'attributs. Même dans le cas où chaque clé est composée de quelques attributs, le nombre de clés candidates dépasse rapidement le million. Par exemple, considérons une relation décrite par 60 attributs le nombre de clés candidates serait de 2^{60} -1. Même si nous savons que toutes les clés existantes sont composées d'au plus cinq propriétés, le nombre de clés candidates est de plus de six millions ($\binom{60}{1}+\binom{60}{2}+\binom{60}{3}+\binom{60}{4}+\binom{60}{5}$). Ainsi, plusieurs méthodes pour élaguer l'espace de recherche ont été proposées.

Clés de bases de données relationnelles. Dans le cadre des bases de données relationnelles, les clés jouent un rôle très important car elles peuvent être utilisées dans des tâches liées à l'intégration de données, la détection d'anomalies, la formulation de requêtes, l'optimisation de requêtes, ou l'indexation. Identifier les clés dans une base de données relationnelle peut également être considéré comme un processus de rétroingénierie des bases de données qui vise à documenter, restructurer ou conserver les données. Par conséquent, plusieurs approches ont proposé des techniques pour les découvrir de façon automatique.

Le problème de la découverte de clés dans une base de données relationnelle a été adressé par diverses approches ([SBHR06], [AN11], [VLM12], [KLL13], [HJAQR$^+$13]). Ces approches peuvent être organisées en deux catégories, les *approches fondées sur des lignes* et *des approches fondées sur les colonnes*. Les approches orientées lignes sont basées sur une analyse ligne par ligne de la base de données pour toutes les combinaisons de colonnes. Toutes approches fondées sur la recherche de colonnes pour la découverte de clés procède colonne par colonne. Pour améliorer l'efficacité de la découverte de clé, des *approches hybrides* utilisant les deux techniques ont également été introduites.

Clés dans le Web Sémantique. Même si le problème de la découverte de clés dans les bases de données relationnelles est similaire à celui du Web Sémantique, différentes caractéristiques des données RDF doivent être prises en compte. Par exemple, les données RDF sont généralement composées de nombreux triplets et de nombreuses propriétés, de nouvelles stratégies d'élagage et de stockage de données, tenant compte de la sémantique de l'ontologie sont nécessaires.

La particularité des données RDF qui contiennent des propriétés à valeurs multiples, rendent inapplicables les approches découvrant des clés dans les bases de données. En outre, les données RDF sont généralement conformes à une ontologie où la connaissance de la hiérarchie de classes existe. Très souvent, les jeux de données RDF contiennent des données incomplètes. Pour faire face à cela, différentes hypothèses pour les informations non déclarées dans les données sont considérées. La plupart des approches proposées, à la fois dans les bases de données relationnelles et le Web Sémantique, considèrent que les données utilisées dans la découverte de clés sont localement complètes. Seulement l'approche [KLL13], présentée dans le cadre des bases de données relationnelles, propose différentes heuristiques de l'interprétation des valeurs NULL dans une table relationnelle.

Le problème de découvrir l'ensemble des clés composites minimales dans un jeu de données RDF a été seulement abordé dans [ADS12]. Néanmoins, dans ce travail, les auteurs fournissent une approche qui découvre des clés non conformes à la sémantique de clés telle

qu' elle est déclarée par OWL2 (voir [ACC+14] pour une étude comparative). Ce travail ne prévoit pas de stratégies d'élagage qui peuvent améliorer l'évolutivité de la découverte. Les autres approches se concentrent uniquement sur la découverte soit de clés simples ou un petit sous-ensemble de clés composites. Toutes ces approches de découverte de clés considèrent que toutes les données sont localement complètes.

Pour estimer la qualité des clés découvertes, différents paramètres tels que le support et la discriminabilité de chaque clé sont utilisés. Aucune des approches existantes ne propose de stratégie pour fusionner les clés découvertes dans différents jeux de données. Une stratégie de fusion peut permettre la découverte de clés avec une qualité supérieure. En effet, des clés valides sur toutes les sources garantissent une meilleure qualité des résultats.

Ainsi, nous considérons que les approches qui découvrent efficacement un ensemble de clés OWL2, en tenant compte des doublons et des données erronées, sont nécessaires. De plus, les données RDF peuvent être incomplètes, différentes hypothèse permettant d'expliquer les données manquantes sont aussi nécessaires.

KD2R: A Key Discovery approach for Data Linking

Nous allons tout d'abord présenter KD2R, une approche automatique de découverte de clés composites pour des jeux de données RDF conformes aux ontologies OWL. Pour découvrir les clés dans des jeux de données où l'hypothèse du Close World Assumption (CWA) n'est pas assurée, nous avons théoriquement besoin de tous les liens *owl:sameAs* et *owl:differentFrom* existants dans un jeu de données. Comme, en général des jeux de données RDF ne contiennent pas ces liens et que l'hypothèse de CWA ne peut pas être assurée, KD2R découvre des clés dans des jeux de données où l'hypothèse de nom unique (UNA) est satisfaite. En d'autres termes, il existe un lien *owl:differentFrom* implicite entre chaque paire d'instances dans les données. En outre, la découverte de clés lorsque les données peuvent être incomplets est également possible. KD2R utilise une heuristique pessimiste ou optimiste afin d'interpréter l'absence d'informations dans les données. Pour être plus efficace, KD2R découvre tout d'abord les *non keys* maximales (c'est à dire, un ensemble de propriétés partageant des communes valeurs pour plusieurs instances distinctes) avant de déduire les clés. De plus, à l'aide de l'ontologie, KD2R exploite l'héritage entre classes afin de couper l'espace de recherche. Afin d'obtenir des clés valides dans différents jeux de données, KD2R découvre en premier des clés valides dans chaque jeu de données, puis applique une opération de fusion. Pour trouver les clés dans des ensembles de données se conformant à des ontologies distinctes, des outils d'alignement d'ontologies sont utilisées. Ces outils décrouvrent des correspondances entre éléments de l'ontologie (voir [PJ13] pour

un état de l'art récent sur l'alignement d'ontologies). Ces correspondances sont exploitées pour trouver les clés qui sont valides dans tous les ensembles de données.

Définitions. Dans cette approche, nous considérons un ensemble de propriétés comme une clé pour une classe, si chaque instance de la classe est uniquement identifiée par cet ensemble de propriétés. En d'autres termes, un ensemble de propriétés est une clé pour une classe si, pour toutes les paires de instances distincts de cette classe, il existe au moins une propriété dans cette ensemble pour lequel toutes les valeurs sont distinctes.

Definition 1. (Key). *Un ensemble de propriétés P ($P \subseteq \mathscr{P}$) est une clé pour la classe c ($c \in \mathscr{C}$) dans un jeu de données D si:*

$$\forall X\ \forall Y\ ((X \neq Y) \wedge c(X) \wedge c(Y)) \Rightarrow$$

$$\exists p_j\ (\exists U\ \exists V\ p_j(X,U) \wedge p_j(Y,V)) \wedge (\forall Z\ \neg(p_j(X,Z) \wedge p_j(Y,Z)))$$

Afin de minimiser le nombre de calculs pour la découverte des clés, nous proposons une méthode inspirée de [SBHR06], qui génère d'abord l'ensemble des *non keys* maximales (c'est à dire, des ensembles de propriétés qui partagent les mêmes valeurs pour au moins deux instances), puis l'ensemble des clés minimales à partir des non keys. Contrairement aux clés, ayant seulement deux instances partageant des valeurs pour un ensemble de propriétés sont suffit de considérer cet ensemble comme non key.

Nous considérons un ensemble de propriétés comme non key pour une classe c s'il existe au moins deux instances distinctes de cette classe qui partagent des valeurs pour toutes les propriétés de cet ensemble.

Definition 2. (Non key). *Un ensemble de propriétés P ($P \subseteq \mathscr{P}$) est une non key pour la classe c ($c \in \mathscr{C}$) dans un jeu de données D si:*

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

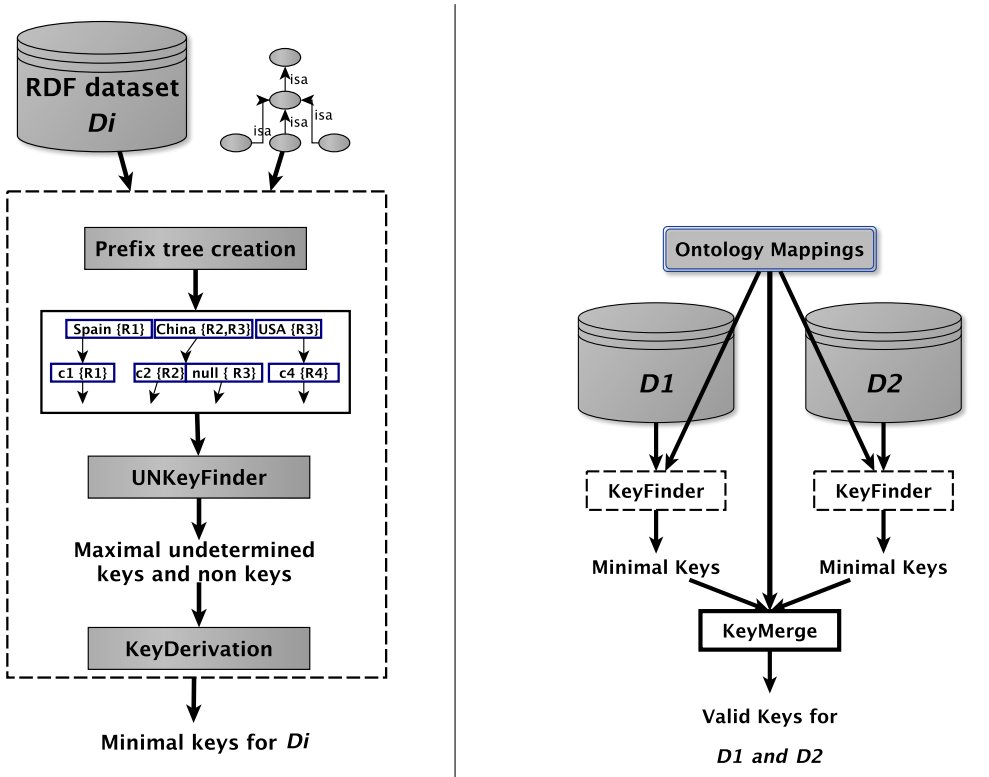
KD2R propose différentes heuristiques pour interpréter l'absence d'informations. Puisque quelques combinaisons de propriétés ne peuvent pas être considérées ni comme des clés ni comme non keys en raison de l'absence éventuelle de données, KD2R introduit la notion des *undetermined keys* qui représentent un ensemble de propriétés pour une classe c où: (i) cet ensemble de propriétés n'est pas une non key et (ii) il existe au moins deux instances de la classe qui partagent des valeurs pour un sous-ensemble de la clé indéterminée et (iii) les

propriétés restantes ne sont pas instanciées pour au moins une des deux instances.

Definition 3. (Undetermined key). *Un ensemble de propriétés P ($P \subseteq \mathscr{P}$) est undetermined key pour la classe c ($c \in \mathscr{C}$) dans un jeu de données D si:*

- *(i)* $P \notin NK_{D.c}$ *et*
- *(ii)* $\exists X \; \exists Y \; (c(X) \wedge c(Y) \wedge (X \neq Y) \wedge \forall p_j$
  $((\exists Z \; (p_j(X,Z) \wedge p_j(Y,Z)) \vee \nexists W \; (p_j(X,W) \vee \nexists W \; p_j(Y,W))))$

Undetermined keys peuvent être considérées soit comme des clés ou comme des non clés, en fonction de l'heuristique sélectionnée. En utilisant l'heuristique pessimiste, les clés indéterminées sont considérés comme des non keys, en utilisant l'heuristique optimiste, elles sont considérées comme des clés. Les clés indéterminées découvertes peuvent également être validées par un expert humain qui peut les affecter à l'ensemble des clés ou des clés non.



(a) KeyFinder pour un jeu de données | (b) Fusion des keys pour deux jeux de données

Fig. 1 Decouverte des clés pour deux jeux de données

Dans la Figure 1, nous montrons les étapes principales de l'approche KD2R. Notre méthode découvre des clés indépendamment pour chaque jeu de données RDF. Dans chaque jeu de données, KD2R est appliquée sur les classes qui sont déjà triées dans un ordre topologique. De cette façon, les clés découvertes dans les super-classes peuvent être exploitées lorsque les clés sont découvertes dans leurs sous-classes. Pour un jeu de données *Di* et une classe donnée *c*, nous appliquons KeyFinder, un algorithme qui trouve des clés pour chaque classe d'un jeu de données. Les instances d'une classe donnée sont représentés dans un préfix-tree (voir Figure 1(a)). Cette structure est utilisée pour découvrir l'ensemble des clés indéterminées maximales et de non keys maximales. Une fois que toutes les clés indéterminées et les non keys sont découvertes, elles sont utilisées pour dériver l'ensemble des clés minimales. KeyFinder répète ce processus pour chaque classe de l'ontologie donnée. Pour calculer les clés qui sont valides pour les classes de deux ontologies, KeyFinder est appliqué dans chaque jeu de données indépendamment et une fois que toutes les clés sont trouvées pour chaque classe, les clés obtenues sont ensuite fusionnées pour calculer l'ensemble des clés qui sont valides dans plusieurs ensembles de données (voir Figure 1(b)).

Pour évaluer notre approche, nous avons exécuté KD2R dans différents ensembles de données RDF. Les expérimentations ont montré que KD2R se comporte bien dans des jeux de données où le nombre de propriétés dans les données est limité. Pour de grands jeux de données, contenant de nombreuses propriétés, KD2R ne peut pas être appliqué. Dans tous les cas, l'heuristique optimiste est plus rapide que l'heuristique pessimiste. Cependant, la dérivation de clés à partir de non keys reste le goulot d'étranglement de KD2R.

Expérimentations. Pour évaluer l'intérêt des clés découvertes, nous avons lié des données en utilisant (i) des clés de KD2R (ii) des clés déclarées par des experts et (iii) aucune clé. Les expérimentations ont montré que lorsque les clés de KD2R sont utilisées, les résultats sont meilleurs que lorsque aucune clé n'est appliquée et sont semblables à ceux utilisant les clés déclarées par des experts. En comparant les clés trouvées avec l'heuristique optimiste et l'heuristique pessimiste, nous avons montré que les clés optimistes conduisent à un meilleur liage pour les jeux de données testés.

SAKey: Scalable Almost Key discovery

Les données publiées sur le Web sont généralement créés automatiquement, donc elles peuvent contenir des informations erronées. En outre, des URI distinctes qui se réfèrent au même objet du monde réel, c'est à dire, des doublons, peuvent exister dans les sources. Considérant qu'une clé unique identifie toutes les instances dans un jeu de données, si les

données utilisées pour la découverte des clés contient des informations erronées ou des doublons, des clés pertinentes peuvent être perdues. Ainsi, les algorithmes qui recherchent uniquement des clés, ne sont pas capable de découvrir toutes les clés dans ces ensembles de données. Pour cette raison, il est essentiel de développer une approche qui permet la découverte de clés malgré la présence de certaines instances qui les violent. Les instances qui mènent à ces violations sont appelées exceptions.

Une caractéristique importante de jeux de données RDF qui sont disponibles sur le Web est leur grand volume. Pour faire face à cela, nous avons d'abord développé KD2R, une approche qui découvre l'ensemble des non keys maximales avant de dériver l'ensemble des clés minimales. Cette ordre rend la découverte des clés plus efficace. Néanmoins, KD2R est submergé par l'énorme quantité de données disponibles sur le Web. Aussi, nous présentons une nouvelle méthode appelée SAKey (Scalable Almost Key discovery), qui est capable de découvrir des clés sur les grands jeux de données malgré la présence d'erreurs et/ou de doublons. Nous appelons les clés découvertes par SAKey, *almost keys*. Une almost key représente un ensemble de propriétés qui n' est pas une clé à cause de quelques exceptions. Comme dans KD2R, l'ensemble des almost keys est dérivé de l'ensemble des non keys trouvé dans les données. SAKey est capable de s'exécuter sur de grands jeux de données. Ce sont ses techniques de filtrage et ses stratégies d'élagage qui réduisent les exigences de temps et espace de la découverte des non keys. Étant donné que la dérivation de clés à partir de non keys est considérée comme le goulot d'étranglement de KD2R, dans SAKey nous avons introduit un nouveau algorithme plus efficace pour la dérivation des clés. Enfin, nous proposons une extension de SAKey pour la découverte de clés conditionnelles. Il s'agit de clés qui sont valides dans des conditions spécifiques. Pour faire face à l'incomplétude des données, SAKey considère que chaque valeur non déclarée dans les données est différente de ce qui existe dans les données. Cette hypothèse correspond à l'heuristique optimiste, introduite par KD2R. Dans nos expérimentations, cette heuristique s'est montrée beaucoup plus rapide et a conduit à de meilleurs résultats que l'heuristique pessimiste.

Définitions. Dans SAKey, nous définissons une nouvelle notion de clés qui permet des exceptions, appelées *almost keys*. Un ensemble de propriétés est une almost key si il existe au plus n instances qui partagent les même valeurs pour un ensemble de propriétés dans le jeu de données considéré.

Formellement, une exception représente une instance qui partages des values de ces propriétés avec au moins une autre instance.

Definition 4. (Exception). *Une instance X de la classe c ($c \in \mathscr{C}$) est une exception pour un*

ensemble des propriétés P ($P \subseteq \mathscr{P}$) si:

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

Un ensemble de propriétés est considéré comme une almost key, s'il existe de 1 à n exceptions dans le jeu de données. En utilisant l'ensemble des exceptions E_P, nous donnons la définition suivante d'une almost key.

Definition 5. (n-almost key). *Soit c une classe ($c \in \mathscr{C}$), P un ensemble des propriétés ($P \subseteq \mathscr{P}$) et n un entier. P est une n-*almost key *pour c si $|E_P| \leq n$.*

Comme nous l'avons vu dans KD2R, un moyen efficace pour obtenir des clés est de découvrir d'abord toutes les non keys et de les utiliser pour calculer les clés. En appliquant cette idée initialement proposée dans [SBHR06], SAKey dérive l'ensemble des almost keys à partir de l'ensemble des propriétés qui sont pas des almost keys. En effet, pour montrer qu'un ensemble de propriétés n'est pas une n-almost key, c'est-à-dire, un ensemble de propriétés avec au plus n exceptions, il suffit de trouver au moins $(n+1)$ instances qui partagent des valeurs pour cet ensemble. Nous appelons les ensembles de propriétés qui ne sont pas almost keys, $(n+1)$*-non keys*.

Definition 6. (n-non key). *Soit c une classe ($c \in \mathscr{C}$), P un ensemble des propriétés ($P \subseteq \mathscr{P}$) et n un entier, P est un n-*non key *pour c si $|E_P| \geq n$.*

L'approche SAKey trouve des almost keys pour un jeu de données RDF et une classe définie dans une ontologie. SAKey est composé de trois étapes principales: (1) les étapes de prétraitement qui nous permettent de filtrer les données et d'éliminer des ensembles de propriétés non pertinents (2) la découverte de (n+1)-non keys maximales en appliquant des stratégies d'élagage et des heuristiques d'ordonnancement et enfin (3) un algorithme qui dérive efficacement des almost keys à partir d'un ensemble de (n+1)-non keys.

Les deux approches [SBHR06, PSS13] dérivent l'ensemble des clés en itérant les deux étapes suivantes: (1) le calcul du produit cartésien des ensembles de propriétés complémentaires aux non keys découvertes et (2) la sélection des ensembles minimaux. La dérivation des clés en utilisant cet algorithme prend beaucoup de temps lorsque le nombre de propriétés est grande. Pour éviter les calculs inutiles, nous proposons un nouvel algorithme, appelé *keyDerivation*, qui dérive rapidement des almost keys minimales. Dans cet algorithme, les propriétés sont ordonnées en utilisant leurs fréquence dans les ensembles de compléments. A chaque itération, la propriété la plus fréquente est sélectionnée et toutes les almost keys

impliquant cette propriété sont découvertes de manière récursive. Pour chaque propriété p sélectionnée, nous combinons p avec les propriétés des ensembles de complément sélectionnés qui ne contiennent pas p. En effet, seulement les ensembles de complément qui ne contiennent pas cette propriété peut conduire à la construction des almost keys minimales. Quand toutes les almost keys contenant p sont découvertes, cette propriété est éliminée de chaque ensemble de complément. Lorsque au moins un ensemble de complément est vide, toutes les almost keys ont été découvertes. Si chaque propriété a une fréquence différente dans les ensembles de complément, toutes les almost keys trouvées sont des almost keys minimales. Dans le cas où deux propriétés ont la même fréquence, des heuristiques supplémentaires sont prises en compte pour éviter les calculs des almost keys non minimales.

Expérimentations. Nos expérimentations approfondies ont montré que SAKey peut fonctionner sur des millions de triplets grâce à ses techniques de filtrage et ses stratégies d'élagage. Le passage à l'échelle de l'approche a été évalué sur des jeux de données différents. Même si de nombreuses exceptions sont autorisées, SAKey peut encore découvrir des n-non keys de manière très efficace. En outre, nous observons que SAKey est beaucoup plus rapide que KD2R tant à la découverte des non keys et à la dérivation des keys à partir des non keys. Enfin, les expérimentations sur l'utilisation des almost keys pour le liage des données montrent que les résultats s'améliorent lorsque quelques exceptions sont autorisées.

C-SAKey. Afin d'enrichir, autant que possible, l'ensemble des keys qui peuvent être déclarées pour un domaine spécifique, nous proposons également C-SAKey, une extension de SAKey qui découvre des *conditional keys*. Un ensemble de propriétés est une conditional key pour une classe, s'il s'agit d'une clé pour les instances de la classe qui satisfont une condition spécifique. Ici, nous considérons des conditions qui impliquent une ou plusieurs propriétés pour lequel une valeur est spécifiée. Plus précisément, étant donné une classe c ($c \in \mathscr{C}$), une instance X et l'ensemble des propriétés $P = \{p_1, \ldots, P_M\}$ où $P \in \mathscr{P}$, une condition $Cond(X)$ peut être exprimée comme:

$$p_1(X, v_1) \wedge \ldots \wedge p_m(X, v_m)$$

Une expérimentation préliminaire pour l'évaluation de C-SAKey a été effectuée et a démontré que les conditional keys peuvent être découvertes pour des ensembles de données pour lesquels aucune clé n'avait pu être trouvée.

Conclusion

Enrichir les connaissances dans le Web sémantique est aujourd'hui un point crucial. Dans cette thèse, nous avons poursuivi cet objectif en étudiant la conception et la mise en œuvre des approches capables de découvrir des clés OWL2 dans les données RDF. Ces données RDF peuvent être nombreuses, incomplètes et contenir des erreurs ou des doublons. Nous avons proposé deux approches, KD2R et SAKey, qui sont capables de découvrir l' ensemble de clés OWL2 dans des jeux de données RDF se conformant à une ontologie OWL. Ces deux approches ont été adoptées pour répondre à différents problèmes.

KD2R peut s'attaquer à des ensembles de données où l'hypothèse UNA est satisfaite. Pour obtenir des clés valables pour différents jeux de données se conformant à des ontologies distinctes, nous découvrons des clés contenant des propriétés alignées trouvées dans chaque jeu de données considéré. Une fois que toutes les clés sont découvertes, nous appliquons une étape de fusion pour trouver l'ensemble des clés minimales qui sont valides dans chaque jeu de données. KD2R prend en compte les caractéristiques des données RDF tels que l'incomplétude et la multi-valuation. KD2R propose deux heuristiques différentes afin de travailler avec des données incomplètes, l'heuristique pessimiste et l'heuristique optimiste. Comme les données peuvent être nombreuses, une stratégie qui découvre d'abord des non keys maximales est utilisée pour calculer les clés. En effet, pour découvrir qu'un ensemble de propriétés est une non key, seulement un sous-ensemble des données est nécessaire. Les expérimentations ont montré que KD2R est plus efficace dans de petits ensembles de données où le nombre de propriétés est limité.

D'autre part, sachant que les erreurs ou des doublons peuvent conduire à la perte de clés, Nous avons aussi introduit SAKey, une approche qui découvre des clés sur de grands ensembles de données RDF contenant des erreurs et des doublons. SAKey découvre des almost keys, c'est-à-dire des jeux de propriétés qui ne sont pas des clés dues à quelques exceptions dans les données. Pour des raisons d'efficacité, SAKey découvre d'abord l'ensemble des (n-1)-non keys maximales et les exploite pour obtenir l'ensemble des almost keys minimales ensuite. Pour passer à l'échelle, SAKey applique une série de techniques de filtrage afin d'écarter une partie des données qui ne peuvent pas conduire à des (n-1)-non keys maximales. En outre, SAKey utilise un certain nombre de stratégies d'élagage qui permet de découvrir rapidement toutes les (n-1)-non keys maximales. En utilisant des heuristiques d'ordonnancement, SAKey parvient à découvrir n-non keys encore plus vite. Contrairement à KD2R, SAKey est capable de traiter de grands jeux de données composés d'un grand nombre de propriétés. Contrairement à l'algorithme de dérivation de clé utilisé dans KD2R, SAKey introduit un nouvel algorithme de dérivation des clés qui est capable de cal-

culer les ensembles des almost keys très efficacement. La dérivation des clés n' est plus le goulot d'étranglement d'approches qui découvrent d'abord des non keys.

Nos expérimentations approfondies ont montré que SAKey peut fonctionner sur des millions de triplets grâce à ses techniques de filtrage et de ses stratégies d'élagage. La validité de l'approche a été montrée validée sur différents jeux de données. Même quand de nombreuses exceptions sont autorisées, SAKey peut encore découvrir efficacement des n-non keys. Nous observons que SAKey est beaucoup plus rapide que KD2R, à la fois pour la dérivation des clés et la découverte des non keys. Enfin, les expérimentations sur le liage de données montrent que les résultats s'améliorent lorsque quelques exceptions sont autorisées. Ainsi, les expérimentations démontrent globalement la validité et la pertinence des clés découvertes. Une expérimentation préliminaire pour évaluer C-SAKey a été effectuée et a démontré que des clés conditionnelles peuvent être découvertes dans des jeux de données où des clés ne peuvent pas être trouvées.

Diverses pistes de travail sont présentées dans cette thèse: (i) une évaluation expérimentale approfondie de SAKey, (ii) un algorithme efficace pour C-SAKey, (iii) une interface graphique pour les experts, (iv) un réglage automatique du nombre d'exceptions n à autoriser, (v) une stratégie pour la fusion des clés, (vi) une méthode pour la construction des fonctions de similarité complexes en utilisant la valeur n, (vii) des métriques différentes pour la qualité de clés, (viii) une approche pour la découverte des clés dans des données RDF hétérogènes, (ix) une approche pour la découverte des clés contenant des chaînes de propriétés, (x) une approche pour la découverte de dépendances sémantiques, (xi) une méthode qui met à jour efficacement des clés lorsque les données évoluent.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and objectives

Over the recent years, the number of RDF datasets available on the Linked Open Data cloud (LOD) has led to an explosive growth of the Web of Data. In July 2014, the number of triples in the LOD has surpassed 61 billions[1].

In this data space, *linking* information from different data sources is a crucial point for real innovation. Once data are connected with other data, by the use of typed links (*semantic links*), they become available in different contexts and thus, more applications and knowledge can be generated. Establishing semantic links between data items represents the ability to transform *local* datasets into a *global data space* (the Web of Data). Semantic links can be generic and they can state, for example, that two data items refer to the same real world object or they can be more "domain dependent" and state, for example, that the city "Paris" is located near the city "Versailles".

Comparing to the huge number of triples available on the Web, the number of existing links between these triples is very small. Indeed, in the statistic cited before, only 643 millions of links exist. Discovering and publishing new links in the LOD is, today, an interesting research topic. In fact, once the data are correctly and significantly linked, they can be consumed by emerging Linked Data-driven applications such as linked data browsers, search engines, crawlers and domain specific linked data applications.

Semantic links can be set manually by a human expert. However, considering the large amount of data available on the Web, the manual creation of semantic links is becoming a not feasible option. Among the different kinds of semantic links that can be established,

[1] http://stats.lod2.eu

identity links, called also sameAs statements, express that different resources refer to the same world entity. For example, an identity link between two instances of people states that these two instances refer to the same person. Today, most of the links between different datasets in the LOD are basically identity links.

In the literature, there exist a lot of approaches that aim to detect identity links between data items. Many approaches use rules to specify the conditions that two data items must fulfill in order to be linked. These rules can be very specific and expressed using different similarity measures, weights and thresholds. This knowledge can be either specified by a human expert or learned from labeled datasets. Other approaches are based on logical rules such as the (inverse) functionality of properties or *keys*. A key represents a set of properties that uniquely identifies each resource. In other words, if two resources share values for all the properties of a key, then they refer to the same real world entity. Nevertheless, when properties and classes are numerous, keys cannot be easily specified by a human expert. Indeed, it is difficult to assume that an expert is always available to specify keys for every dataset and every application domain. Moreover, this task becomes even harder when a complete set of composite keys is needed. When the data are incomplete, the more keys are numerous and involve different properties, the more decisions can be taken in the data linking process. The problem of discovering the complete set of keys automatically from the data is #P-hard [GKM$^+$03]. Therefore, approaches able to discover keys *efficiently* are essential. Additionally, since the data available on the Web are built in an autonomous way and conform to different application domains, the datasets are, by construction, *heterogeneous* and may contain errors and duplicates. In this case, the key discovery problem becomes much more complex. Furthermore, in the Semantic Web context, RDF data may be incomplete and asserting the Closed World Assumption (stating that what is not known to be true is false) may not be meaningful. Hence, novel strategies that discover keys in big RDF data that may be dirty and incomplete are required.

In this thesis, we study the theoretical and practical aspects of the automatic key discovery in RDF data. More precisely, we aim at developing key discovery approaches that are able to handle datasets where the data can be numerous, incomplete and erroneous. The objective is to discover as many keys as possible, even ones that do not apply to the whole data.

1.2 Contributions

We have developed different approaches for the automatic discovery of keys in RDF data. These approaches discover sets of composite keys in several datasets, each one conforming to an OWL ontology. The discovered keys follow the semantics of the *owl:HasKey* construct of OWL2. Two different approaches are proposed:

1. KD2R is an approach that discovers OWL2 keys from the data, using either a pessimistic or an optimistic heuristic to interpret the absence of information.

2. SAKey is an approach that discovers OWL2 keys from the data containing erroneous information or duplicates by ensuring good scalability characteristics.

KD2R: Key Discovery approach for data linking.
Our first contribution is KD2R, an approach that discovers automatically the complete set of composite keys in several RDF datasets. More precisely, since every superset of a key is a key, KD2R is only interested in the discovery of minimal keys. KD2R finds first keys for each class of a RDF dataset. The classes are exploited from the more generic to the most specific ones. Once all the keys per class and per dataset are found, KD2R uses a merge operation to obtain, for each equivalent class, keys that are valid in every dataset.

To avoid losing keys due to duplicates, KD2R considers that all the instances of a dataset are distinct (Unique Name Assumption). Since RDF data are usually incomplete, KD2R uses two different heuristics, the optimistic heuristic and the pessimistic heuristic, to interpret the absence of information in a dataset. In the optimistic heuristic, the property values that are not declared in a dataset, are assumed to be different from all the existing values in this dataset. In the pessimistic heuristic, when a property is not used in the description of an instance, this property can take any of values that appear in the dataset.

To ensure the efficiency of key discovery, KD2R discovers first all the sets of properties that are not keys, i.e., *non keys*. More precisely, KD2R discovers the complete set of maximal non keys, representing sets of properties that by adding one property become keys. In this way, all the sets of properties that are not included or equal to a non key refer to keys. To improve even more the efficiency of key discovery, KD2R applies different pruning strategies such as the key inheritance.

A extensive experimental evaluation of KD2R has been conducted to evaluate first the scalability of the approach and second the impact of keys in the data linking task. Pessimistic and optimistic heuristics are also compared. According to the experiments, KD2R performs well in datasets containing classes described by few properties and that

KD2R is more scalable when the optimistic heuristic is applied. Moreover, the optimistic heuristic appears to have better results in the linking task than the pessimistic heuristic. Finally, the data linking results have demonstrated that the use of KD2R keys gives (i) better linking results than those obtained by applying no keys to link and (ii) similar results to those obtained by applying expert keys to link.

SAKey: Scalable Almost Key discovery

Our second contribution is an approach that discovers efficiently keys in incomplete RDF data that may contain erroneous information and duplicates. Thus, it is important to be able to discover sets of properties that might not be keys due to few exceptions. An exception for a given key represents a data item that shares values with another data item for this set of properties. To allow the discovery of keys with exceptions we propose SAKey, an approach that discovers keys that almost hold, called *almost keys*, in one dataset. The number of allowed exceptions is parametrized by fixing a value n. Then, SAKey discovers n-almost keys, i.e., sets of properties that share values for at most n instances. Each of these n instances represents one exception. Apart from real keys that can be lost due to erroneous data or duplicates, almost keys represent also sets of properties with a high linking power.

In SAKey, to discover keys under the Open World Assumption, we consider that a heuristic to interpret the absence of information is applied. SAKey applies the optimistic heuristic since better linking results have been obtained using it in KD2R.

Similarly to KD2R, SAKey discovers first maximal sets of properties that are not almost keys, i.e., n-non keys, and then uses them to derive the complete set of minimal n-almost keys. To be scalable in the settings of the Semantic Web where datasets can be composed of millions of triples, SAKey applies a series of filtering techniques and pruning strategies to discover efficiently the set of properties that are n-non keys. In particular, semantic dependencies found during the n-non key discovery process are exploited to prune the search space.

Once the sets of properties that are not almost keys are discovered, all the minimal n-almost keys are derived using a new very efficient algorithm. This algorithm uses the frequencies of properties to exploit first properties that can lead very fast to the derivation of minimal n-almost keys. A merge operation similar to the one proposed in KD2R is introduced in order to obtain valid n-almost keys discovered in different datasets.

In order to enrich, as much as possible, the sets of discovered keys, we propose also C-SAKey, an extension of SAKey that discovers *conditional keys*. A conditional key is a key that is valid only in a subpart of the data. This part of the data is defined using a condition

in the form "property = value".

SAKey has been tested in several RDF datasets. The experimental evaluation is threefold. First, to show the impact of filtering techniques and pruning strategies in the scalability of the approach. Second, to compare the efficiency of the discovery of keys in KD2R and SAKey both for the discovery of non keys and the derivation of keys. Finally, to show the impact of n-almost keys in the data linking. The experiments have highlighted that thanks to the filtering techniques SAKey filters out a significant number of data. Pruning strategies allow SAKey to discover n-non keys very efficiently. Comparing to KD2R, SAKey is always orders of magnitude faster, both in the discovery of n-non keys and the derivation of n-almost keys. Finally, the experiments have shown that using n-almost keys in the data linking can significantly improve the quality of the results. To evaluate the interest of conditional keys, one preliminary experiment has been conducted. This experiment shows conditional keys can be discovered in datasets where keys cannot be found.

1.3 Thesis Outline

Below we provide an overview of how the thesis is organized, along with the main contributions of each Chapter.

Chapter 2 provides the necessary background on the RDF data model, on OWL language and on Linked data to follow the rest of the thesis. In the second part of this chapter, the state-of-the-art approaches are described to position the work presented in this thesis. Different approaches that can be used to link data are introduced. We show that some of these approaches use keys to link. Then, we present related works that focus on the discovery of keys or related constraints, in the areas of relational databases and Semantic Web.

Chapter 3 presents KD2R, the first approach that we have developed, to discover OWL2 keys automatically from RDF data that may be incomplete. We first define formally the problem that we tackle in KD2R. Then, we present the algorithms that are used to discover non keys and derive keys from non keys. Finally, we provide an extensive experimental evaluation of KD2R.

Chapter 4 describes SAKey, the approach that discovers keys that almost hold, in incomplete data under the presence of errors and duplicates. First, we formally define the problem of almost key discovery. Then, we provide a series of filtering techniques and pruning strategies used to discover n-non keys. An efficient algorithm for the derivation of n-almost keys from n-non keys is also provided. C-SAKey, the approach that discovers conditional keys is then introduced. Finally, we present the experimentations that have been conducted to evaluate the efficiency and effectiveness of SAKey. Comparative experimental results between SAKey and KD2R are also provided.

Chapter 5 provides a conclusion and discusses various proposals for future work.

Chapter 2

Background and state-of-the-art

This chapter provides first some preliminaries that are needed in this thesis. Then, related works that have studied the problems of data linking and key discovery are presented and discussed.

2.1 Preliminaries

In this section, we present the background of this work. More precisely, Sections 2.1.1 and 2.1.2 introduce the RDF data model and the OWL language. Then, the notion of Linked data is described in Section 2.1.3.

2.1.1 RDF data model

The *Resource Description Framework (RDF)* is a graph data model proposed by W3C for standard model for data representation and interchange on the Web. A resource is anything that can refer to a web page, a location a person a concept etc. A resource is described by a set of triples in the form $< s,\ p,\ o >$ where the subject s denotes a resource that has a value o for the property p. For example, the triple $< p1,\ firstName,\ ''Martin'' >$ states that the resource $p1$ has as first name $''Martin''$. These triples can also be called RDF statements and expressed logically as binary predicates $p(s,\ o)$. In this case, $p(s,\ o)$ is called an RDF fact. Each resource is identified by a *Uniform Resource Identifier (URI)*. The subject and the property of a triple are URIs. The object can be either a URI or a literal value. For example, in the triple $< p1,\ lastName,\ ''Daglas'' >$ the object $''Daglas''$ represents a literal while in the triple $< p1,\ bornIn,\ city1 > city1$ is a URI.

Let U be a set of URIs, L be a set of literals (strings) and B be the set of blank nodes

(anonymous resources).

Definition 1. *(RDF dataset).* An RDF dataset is a set of triples $< s,\ p,\ o >$ where $s \in (U \cup B), p \in U$ and $o \in U \cup B \cup L$.

A collection of RDF triples can be represented as a labeled, directed multi-graph. In this multi graph a node represents a resource, a literal or a blank node while an edge represents a property. This graph structure can be easily extended by new knowledge about an identified resource.

In Figure 2.1, we give an example of RDF data in the form of RDF facts.

Dataset $D1$**:**
$FirstName(p1, "Martin"), LastName(p1, "Johnson"), BornIn(p1, City1),$
$DateOfBirth(p1, "15/03/83"), FirstName(p2, "John"),$
$LastName(p2, "Daglas"), BornIn(p2, City2), DateOfBirth(p2, "16/05/74"),$
$FirstName(p3, "John"), LastName(p3, "Smith"), BornIn(p3, City3),$
$FirstName(p4,"Vanessa"), LastName(p4, "Green"), BornIn(p4, City3)$

Fig. 2.1 An example of RDF data

2.1.2 OWL language

Given resources described in RDF, it is not easy to understand what these resources correspond to and how they can be used. To solve this limitation, descriptions of resources can be related to a vocabulary described in an ontology. An ontology is a formal description that provides users and applications a common understanding for a given domain. It has been defined by Tom Gruber in 1993 [Gru93] as *"a specification of a conceptualization"*. Ontologies allow to organize the data, improve the search, enable reasoning on RDF statements and facilitate data integration. The *Web Ontology Language (OWL)* is a W3C standard for encoding ontologies. Using OWL, classes and properties can be declared in an ontology in a hierarchy using the subsumption relation.

An ontology can be represented as a tuple $\mathscr{O} = (\mathscr{C}, \mathscr{P}, \mathscr{A})$ where:

- $\mathscr{C}$ represents the set of classes,
- $\mathscr{P}$ the set of typed properties for which two kinds of properties are distinguished in OWL2: *datatype properties* where the domain is a class and the range is a literal; and *object properties* where both domain and range are classes.

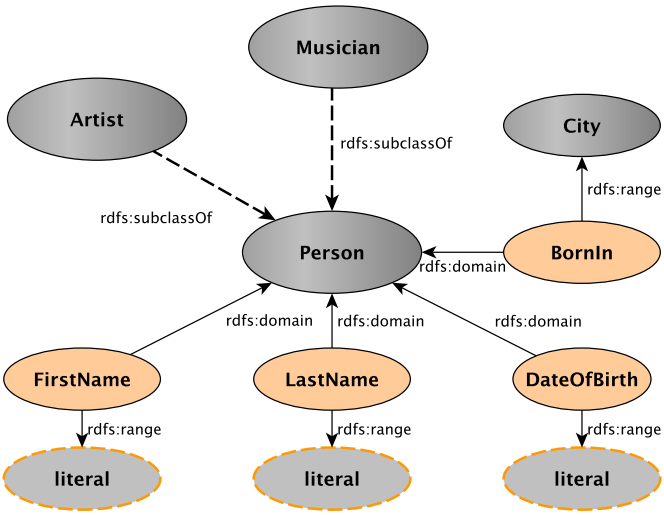


Fig. 2.2 *Person* Ontology

- $\mathscr{A}$ the set of axioms such as subsumption relations between classes or properties, disjunction relations between classes or *owl:HasKey* axioms.

In the sequel, we will use the following notation: $c(i)$ indicates that the resource i belongs to a class c, meaning that i is an *instance* of c.

In Figure 2.2, we present an ontology concerning persons. Each *Person* is described by the datatype properties $FirstName$, $LastName$, $DateOfBirth$ and the object property $BornIn$ which represents the city where as person is born. This ontology contains also the classes $Artist$ and $Musician$, two subclasses of the class *Person*.

A wide variety of constructs are supported by OWL. We only present here constructs that are exploited in related works or in the approaches that we propose in this thesis. First, the construct *owl:FunctionalProperty* is defined by OWL to declare that a property can have only one unique value for each instance. For example, the property $BirthDate$ is functional since a person cannot have more than one date of birth. Moreover, OWL uses the construct *owl:InverseFunctionalProperty* that states that the object of a property uniquely identifies every subject using this property. For example, if the property $BirthDate$ is inverse functional, this means that two persons that share the same date of birth are the same.

In *OWL2*, the new W3C standard for ontology representation which extends OWL, the construct *owl:HasKey* is proposed. This construct is used in a OWL2 ontology to declare that a set of properties $\{p_1,\ldots,p_n\}$ is a key for a given class. It is expressed as *owl:HasKey* $(CE(ope_1,\ldots,ope_m)\ (dpe_1,\ldots,dpe_n))$ which states that each instance of the class expres-

sion CE[1] 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 datatype property expressions dpe_j. An *object property expression* is either an *object property* or an inverse *object property*. More formally,

$$\forall X, \forall Y, \forall Z_1, \ldots, Z_n, \forall T_1, \ldots, T_m \wedge ce(X) \wedge ce(Y) \bigwedge_{i=1}^{n} (ope_i(X, Z_i) \wedge ope_i(Y, Z_i))$$

$$\bigwedge_{i=1}^{m} (dpe_i(X, T_i) \wedge dpe_i(Y, T_i)) \Rightarrow X = Y$$

where $ce(X)$ denotes that the X is an instance of the class expression ce. The only allowed *datatype property expression* is a *datatype property*. For example, we can express that the property expression {*LastName*, *BirthDate*} is a key for the class *Person* using *owl:HasKey(Person (() (LastName*, *BirthDate*))*.*

Note that, a functional object property is a single key for all the resources that appear as objects of this property while inverse functional (data type and object) properties are single keys for all the resources that appear as subjects of this property.

2.1.3 Linked data

Typed links between resources described in different datasets can be declared to allow users, crawlers or applications navigate along links and combine information from different datasets. For example, an RDF triple can state that a person $p1$ described in one dataset is the author of the article $a1$ found in another dataset. One of the most important links, the identity link, states that two URIs refer to the same real world object. In OWL, an identity link between two instances is declared using the construct *owl:sameAs*. Linked data [BL06] has been introduced by Tim Berners-Lee in 2006 to refer to best practices for publishing and interlinking such structured data on the Web. A significant number of organizations use these practices to publish their data in the Web. The result is the Linked Open Data Cloud (LOD) (see Figure 2.3), a global graph containing a big number of RDF triples. In July 2014, LOD contained more than 61 billions of RDF triples while only 643 millions of links[2].

[1] We consider only the class expressions that represent OWL classes

[2] http://stats.lod2.eu

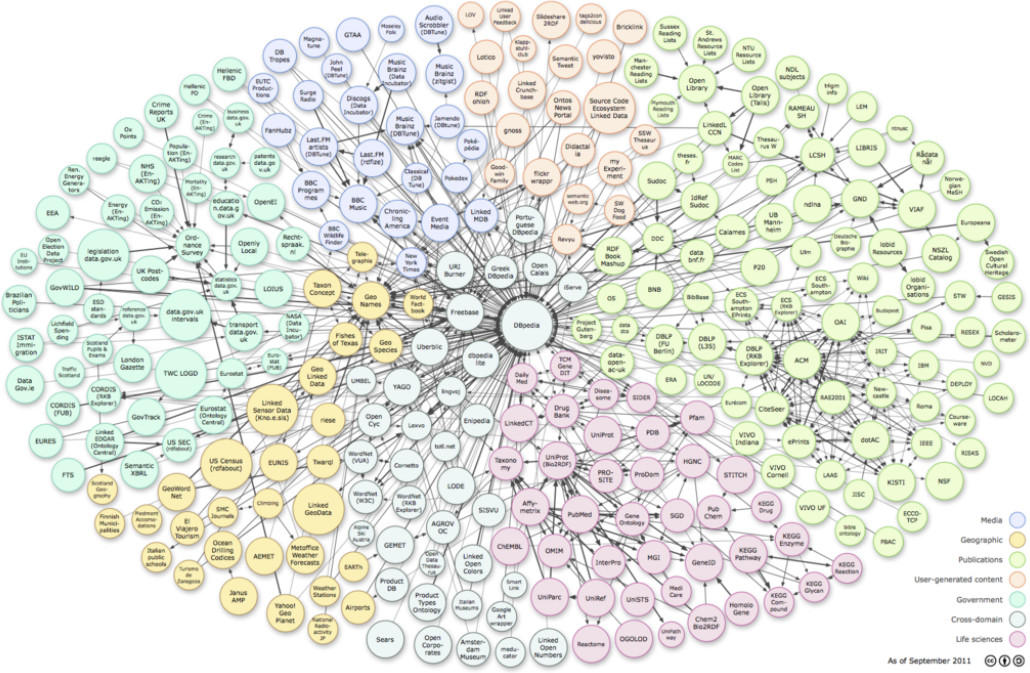


Fig. 2.3 Linked Open Data Cloud Diagram - September 2011 [3]

2.2 State-of-the-art

In this section, we introduce first, approaches that focus on the data linking problem using rules. Then we present approaches that have been proposed for the discovery of keys and functional dependencies in the setting of relational databases and Semantic Web.

2.2.1 Data Linking

Considering the large amount of data available on the Web, only few links between datasets exist. Many approaches propose methods that generate these links between RDF datasets automatically. In Semantic Web, the task of generating links is usually referred as *data linking*, *instance matching*, *interlinking* or *reference reconciliation*. Data linking can be seen as an operation that takes as input two datasets, each one containing a set of descriptions of instances and gives as output links between instances (i.e., *owl:sameAs* links).

The problem of data linking has been known for more than five decades. It has been introduced by [NKAJ59] and first formalized by [FS69]. Many approaches have been proposed in relational databases where the problem is referred as *record linkage*, *data cleaning* or *entity resolution* (see [EIV07] for a survey). Many approaches have been also proposed

[3] http://en.wikipedia.org/wiki/Linked_data

in order to link data in the setting of the Semantic Web. These approaches have been classified in different ways (see [FNS11] for a survey). First, an approach can be considered as supervised or unsupervised. In some supervised approaches a user is needed to control the process of linking. For instance, in [BLGS05], the tool D-Dupe allows a user to choose similarity measures and to validate candidate links via a user interface. Other supervised approaches exploit training data in order to "learn" how to match instances [NL12, IB12]. Unsupervised approaches usually use knowledge that is either declared in an ontology [SPR09, HCQ11] or given by a domain expert [VBGK09, NA11]. Second, an approach can be considered as local (instance-based) or global (graph-based). In local approaches [VBGK09, NA11, IB12, NL12, NdM12], each pair of instances are explored independently while in global approaches [SPR09, HCQ11], discovered links affect the discovery of other links. Moreover, approaches use logical rules to discover high-quality links while other approaches are numerical and compute similarity scores between pairs of instances. Additionally, there exist approaches that use external resources, such as WordNet[4], to improve the data linking [SLZ09, SPR09]. In the thesis, we are interested in approaches that use either logical or numerical rules, manually defined or automatically discovered to link data.

2.2.1.1 Rules in data linking

Rule-based approaches use rules to match instances. A rule is of the form: *If* <*condition*> *Then* <*action*>. A rule is usually composed of complex functions in both *condition* and *action*. The rules can be logical. For example, the logical rule $SSN(p1,y) \wedge SSN(p2,y) \Rightarrow sameAs(p1,p2)$ states that if two people have exactly the same Social Security Number (SSN), then they refer to the same real world person. A rule can be much more complex and be expressed using similarity measures (for more details see [CRF03]), aggregation functions and thresholds. For example, given the descriptions of two museums having names $n1$ and $n2$ and located in cities $c1$ and $c2$ a numerical rule states that if the average of the similarity of names $sim(n1,n2)$ and cities $sim(c1,c2)$ is bigger than 0.9 out of 1, then the two museums are considered as equal.

Logical rules in data linking

Some approaches use knowledge provided by an expert or declared in an ontology to build logical rules for data linking.

[4] http://wordnet.princeton.edu/

The approach ObjectCoref [HCQ11] aims to discover *owl:sameAs* links for one real world object. For example, this approach aims to find instances that refer to the city of Beijing. Given a URI as input, ObjectCoref first builds a *kernel* that contains a set of URIs for which a *owl:sameAs* link is inferred. To infer these links, ObjectCoref exploits semantic knowledge declared in the ontology such as *owl:InverseFunctionalProperty owl:FunctionalProperty*, *owl:cardinality* and *owl:maxCardinality*. Then, a set of unlabeled data is considered. The approach iteratively learns the most *discriminative property-value pairs*, i.e., values that are commonly used in every description and rarely used in the set of unlabeled data. These pairs are used to identify potential *owl:sameAs* links that can be added to the kernel. After a finite number of repetitions, set by an expert, the process finishes.

LN2R is a data linking approach [SPR09] that aims to discover *owl:sameAs* and *owl:differentFrom* links. More precisely LN2R proposes two different methods, L2R that infers *owl:sameAs* and *owl:differentFrom* links using logical rules, and N2R that computes similarity scores between pairs of instances in order to discover *owl:sameAs* links. Both methods exploit functional properties, inverse functional properties, keys and disjunction between classes declared in the ontology. They are exploited by the logical method (L2R) to generate a set of logical inference rules and by the numerical method (N2R) to generate a set of similarity functions. L2R uses logical rules to infer exact decisions of both *owl:sameAs* and *owl:differentFrom* links. Unlike L2R, N2R uses keys to create functions that compute similarity scores for pairs of instances. These functions are expressed using a non linear equation system that is resolved using an iterative method.

Some approaches that are called blocking approaches are based on rules to reduce the number of instance pairs that have to be compared by data linking tools. In [SH11], the authors propose a approach that learns discriminative sets of datatype properties and uses them to create blocks of instances that possibly refer to the same real world entity. For example, the set of properties $\{firstName, LastName\}$ can be used to create blocks of instances of the class *person*. A pair of instances is selected if the similarities of the values of discriminative properties are greater than a given threshold. This kind of approaches can be exploited by data linking tools to pre-process the data. Indeed, more complex rules will only be used on some pairs of instances.

Complex rules in data linking

Since the knowledge that is used to construct linking rules is rarely available in an ontology, some approaches learn the matching rules directly from the data.

In [VBGK09], the authors propose Silk, a tool that discovers identity links between

datasets found on the Web using rules defined by an expert. Silk provides a declarative language for specifying which conditions data items must fulfill in order to be interlinked. Silk uses different aggregation functions, such as MIN, MAX, (weighted) AVERAGE, combined with elementary similarity measures between values to compute a similarity score between two instances. An *owl:sameAs* link is set between two instances when the similarity score of these two instances in the condition of a rule is higher than a given threshold. An example of the specification file is given below.

```
<Silk
 <Prefixes>...</Prefixes>
 <DataSources>
    <DataSource id="dbpedia">...</DataSource>
    <DataSource id="geonames">...</DataSource>
  </DataSources>
  <Interlinks>
    <Interlink id="cities">
      <LinkType>owl:sameAs</LinkType>
      <SourceDataset dataSource="dbpedia" var="a">...</SourceDataset>
      <TargetDataset dataSource="geonames" var="b">...</TargetDataset>
      <LinkageRule>
        <Aggregate type="average">
          <Compare metric="levenshteinDistance" threshold="1">
            <Input path="?a/rdfs:label" />
            <Input path="?b/gn:name" />
          </Compare>
          <Compare metric="num" threshold="1000" >
            <Input path="?a/dbpedia:populationTotal" />
            <Input path="?b/gn:population" />
          </Compare>
        </Aggregate>
      </LinkageRule>
      <Outputs>
        <Output type="file" minConfidence="0.95">
          <Param name="file" value="accepted_links.nt" />
          <Param name="format" value="ntriples" />
        </Output>
      </Outputs>
    </Interlink>
  </Interlinks>
</Silk>
```

In this example, instances of the class city coming from the *sourceDataset* DBpedia and the *targetDataset* Geonames are compared to find *owl:sameAs* links. The expert declares in this specification file that pairs of cities are compared using their names and their populations. Names are compared using *levenshtein* while population using a similarity measure proposed by Silk for numerical values. Only cities for which the average similarity of these two properties is greater than 0.95 are considered as the same.

For scalability reasons, an expert can also specify a declared pre-matching step to select subsets of instances that can refer to *owl:sameAs* links. In this way, this approach manages to prune the search space by avoiding comparing every pair of instances. Silk framework is available on the Web at *https://code.google.com/p/silk*.

Like Silk, LIMES [NA11] is a tool that links data using rules declared by an expert in a specification file. The allowed aggregation functions are MAX, MIN, (weighted) AVERAGE. The originality of LIMES is that this approach uses mathematical characteristics of metric spaces to estimate the similarity between instances and filter out instance pairs. Because of this step, some of the elementary similarity measures like *Jarowinkler* cannot be applied.

The GenLink algorithm introduced in [IB12], is a supervised algorithm that learns expressive linking rules from a set of existing identity links, using genetic programming. Initially, random linking rules involving two properties and randomly chosen similarity measures and aggregation functions are constructed. The results of the data linking using these rules are validated against the identity links to improve the quality of linking rules.

In [NL12], the authors propose EAGLE, an approach based on LIMES [NA11], that combines the use of genetic programming and active learning to discover linking rules in order to link the data. These rules are complex and expressed using simple similarity measures and aggregation functions. The advantage of such approaches is that exploiting active learning techniques they require only a small number of highly informative training data to build the rules. In highly informative training data, each selected link leads to extraction of different knowledge that is used to construct the rules. Thanks to the active learning, EAGLE needs only a small number of training data to construct high quality identity links.

The approach proposed in [NdM12] discovers expressive linking rules which specify the conditions that two data items must fulfill to be linked: data transformations, similarity measures, thresholds and the aggregation function. These rules are learnt using datasets where UNA is fulfilled and a strong degree of overlap exists, applying genetic programming techniques. The data linking is done by applying to the data these linking rules.

2.2.2 Discovering keys

The problem of key discovery has been previously studied in the relational databases. In this setting, keys play a very significant role since they can be used in tasks related to data integration, anomaly detection, query formulation, query optimization, or indexing. Identifying the keys holding in a relational database can be also seen as a part of a database reverse engineering process that aims to document, restructure or maintain the data. Therefore, several approaches have proposed techniques to discover them from the data in an automatic way.

The problem of key discovery is a subproblem of the functional dependency discovery. In the field of relational databases, an active research has been also conducted for the discovery of functional dependencies. Therefore, in this state-of-the-art we present both functional dependency and key discovery approaches.

In the Semantic Web, the interest of key discovery grows more and more thanks to the linked data initiative and to the use of keys in the data linking process. In this state-of-the-art, we present the few approaches that propose strategies for the discovery of keys in RDF data. Furthermore, we also introduce an approach that aims to discover semantic dependencies, a problem that is analogous to the functional dependency discovery in the setting of Semantic Web.

2.2.2.1 Discovering keys and FDs in relational databases

We first introduce the background of relational databases. Then, we present approaches that focus on the discovery of functional dependencies. Approaches that discover *conditional functional dependencies* are also shown. Finally, key discovery approaches are described.

Preliminaries in relational databases

In this section we present the basic notions of relational databases that are needed in the following sections.

Definition 2. *(Relational database).* A relation R is a finite set of attributes $\{A_1, .., A_n\}$. The domain of an attribute A, denoted by $Dom(A)$, is the set of all possible values of A. A tuple is a member of the Cartesian product $Dom(A_1) \times ... \times Dom(A_n)$. A *relational database* is a set of relations R.

A functional dependency states that the value of an attribute is uniquely determined by the values of some other attributes.

Definition 3. *(Functional dependency).* A *functional dependency (FD)* over a relation R is a statement $X \rightarrow A$ where $X \subseteq R$ and $A \in R$. A FD $X \rightarrow A$ is satisfied if whenever two tuples have equal values for X, they also have equal values for A. X corresponds to the left-hand size of the FD while A to the right-hand size.

If $X \rightarrow A$ holds also $Z \rightarrow A$ will hold, for every $X \subset Z$. A functional dependency $X \rightarrow A$ can be defined as minimal when there does not exist $Z \subset X$ for which $Z \rightarrow A$. The set of minimal functional dependencies is a compressed representation of all the functional dependencies that hold in a database.

Given a collection of tuples of a relation, a key is a set of attributes, whose values uniquely identify each tuple in the relation.

Definition 4. *(Key).* A set of attributes $K \subseteq R$ is a *key* for R if K determines all other attributes of the relation, i.e., $K \rightarrow R$.

A key K is minimal if there does not exist a set of attributes K' such that $K' \subset K$ and K' is a key.

Functional dependency discovery in relational databases

The problem of key discovery is a sub-problem of Functional Dependency (FD) discovery in relational databases, since a key determines functionally the remaining attributes within a table. FDs represent semantic constraints within data and can be applied in several domains such as query optimization, normalization and data consistency. FDs were traditionally considered to be given by a human expert. However, an expert may be able to define only a part of the FDs. The discovery of functional dependencies from data has been extensively studied ([SF93], [NC01], [FS99], [HKPT99], [LPL00], [WGR01], [YHB02], [WDS$^+$09]). Such approaches can be used to discover unknown FDs of high importance. For example, in the case of medical data, it might be discovered that carcinogenicity depends functionally from a set of attributes. Since the problem of discovering the complete set of FDs for a given relation is #P-hard [MR94], each approach performs different strategies to minimize, as much as possible, the necessary computations. The proposed approaches can be grouped in two main categories, the *top-down approaches* and *the bottom-up approaches*.

Top-down approaches. In a top-down approach, candidate FDs are generated and then tested against a given relation R. More precisely, the idea of top-down approaches is to start from the set of the most general candidate FDs in the data and then proceed to more

specific ones. A FD $X \rightarrow A$ is more general than a FD $Z \rightarrow A$ when $X \subset Z$. Top-down approaches try to discover only the set of most general FDs, i.e., minimal FDs, since with this set, all the other FDs can be inferred. For example, if the FD $X \rightarrow A$ is discovered, then every candidate $Z \rightarrow A$ with $X \subset Z$ is , by definition, a FD. These approaches are also called levelwise approaches.

In the following, we briefly present approaches proposed in [HKPT99], [NC01] and [YHB02] that are considered as top-down approaches. Each one proposes different pruning techniques to make the FD discovery more efficient.

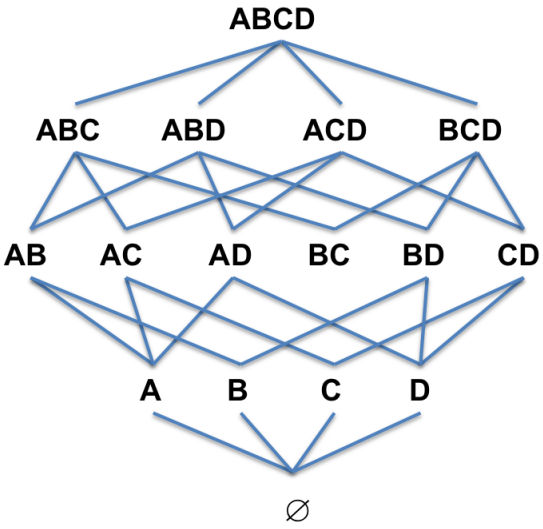


Fig. 2.4 Containment lattice for the relation $R_1(A,B,C,D)$

TANE [HKPT99] is an approach that discovers minimal FDs. All the possible sets of attributes of a relation R_1 can be represented in containment lattice. For example, Figure 2.4 represents the lattice of a relation $R = \{A,B,C,D\}$. Each node in the lattice represents a distinct set of attributes and an edge exists between two nodes X and Y if $Y \subset X$ and X has exactly one more attribute than Y, i.e., $X = Y \cup \{A\}$. TANE uses a compact way to represent the tuples of a relation that is called *stripped partition database*. For each attribute of a relation, the tuples are partitioned according to their common values. Given a value, the set of tuples having this value for an attribute A is called *equivalent class*. The set of equivalent classes for an attribute A is called *partition* and is denoted as $\pi_{\{A\}}$. A stripped partition contains only equivalence classes of size bigger than 1. This means that all the values that uniquely identify a tuple are eliminated since they cannot break any FD. Let us consider the tuples of the relation R_1, given in the Table 2.2. In this example the stripped partition

database is given in the Table 2.1:

$\pi_{\{A\}} = \{\{t1,t2,t3\}\}$
$\pi_{\{B\}} = \{\{t1,t3\}\}$
$\pi_{\{C\}} = \{\{t2,t3\}\}$
$\pi_{\{D\}} = \{\}$

Table 2.1 Stripped equivalent database for the data of the relation R_1 in Table 2.2

This filtering allows TANE to perform faster since it exploits only a subpart of the data in a relation to discover the FDs. TANE searches for FDs of the form: $X\backslash\{A\} \rightarrow A$ where $A \in X$, by examining attribute sets X of increasing sizes: initially sets of size 1, then sets of size 2 and so on. Each X represents a node of the lattice. For each candidate set of attributes to be checked, TANE constructs a list with all the attributes that can lead to a minimal FD. To check if a FD $X\backslash\{A\} \rightarrow A$ is valid, TANE states that the number of equivalent classes of X and $Y = X \cup A$ should be equal ($|\pi_Y(R)| = |\pi_X(R)|$). For example, given the relation R_1 of the Table 2.2, TANE will first level of the lattice that contains single attributes. Using $X\backslash\{A\} \rightarrow A$, TANE explores first the FDs $\emptyset \rightarrow A$, $\emptyset \rightarrow B$, $\emptyset \rightarrow C$, and $\emptyset \rightarrow D$. When all these FDs are explored, TANE continues with the next level, where the FDs to check are now the following: $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $B \rightarrow A$, $B \rightarrow C$, $B \rightarrow D$, $C \rightarrow A$, $C \rightarrow B$, $C \rightarrow D$, $D \rightarrow A$, $D \rightarrow B$ and $D \rightarrow C$.

TANE uses several prunings to improve the scalability of the FD discovery. Already discovered FDs are exploited to prune the search space of candidate FDs. More precisely, TANE states that if $X \rightarrow A$ then $X \cup A \rightarrow Z$ should not be checked since $X \cup A \rightarrow Z$ is not minimal. Moreover, starting from candidate FDs of the form $\emptyset \rightarrow A$, TANE is capable to discover single keys, i.e., attributes that uniquely identify each tuple in a relation. An attribute is a single key when every partition of this attribute has size one. Exploiting the monotonic characteristic of keys, every superset of this attribute will be also a key, therefore, none of these supersets will be explored.

In the previous example, starting from FDs of the first level, only $\emptyset \rightarrow D$ is true since D contains only partitions of size one, therefore it refers to a key. Applying the monotonic pruning, TANE filters out D therefore in the next level TANE explores the FDs $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow A$, $B \rightarrow C$ and $C \rightarrow B$. In the case of $B \rightarrow A$ is true since $|\pi_B(R)| = |\pi_{AB}(R)| = \{t1,t2\}$.

In addition, TANE discovers functional dependencies that almost hold, i.e., approximate FDs. An approximate functional dependency often represents a real functional depen-

dency that cannot be discovered due to erroneous tuples in the data. For example, the FD *SSN* $\rightarrow$ *Name* cannot be discovered if there exist typographical errors in some Social Security Numbers. Moreover, some FDs are almost true, for example the first name of a person determines the gender in most of the cases. TANE considers that $X \rightarrow A$ is an approximate dependency if the number of tuples that have to be removed from the relation R for $X \rightarrow A$ to hold in R is smaller than a given threshold ε.

FUN [NC01] is an approach that exploits *free sets* to discover efficiently FDs. FUN discovers the FDs level by level and exploits the already discovered FDs to obtain directly only minimal ones. Free sets are used to create minimal left-hand sides of FDs. A free set X is a set of attributes for which $\nexists Y \subset X$ such that $|\pi_Y(R)| = |\pi_X(R)|$. A non free set cannot lead to a minimal FD. For example, if for a relation R it is true that $|\pi_A(R)| = |\pi_{AB}(R)|$, this means that no new minimal FD can be found having as left-hand side the attribute set AB. Thus, the main pruning of this work is based on the anti-monotonic behavior of a free set and the monotonic behavior of a non free set. More precisely, every subset of a free set is a free set as well and every superset of a non free set is also a non free set. New FDs can be produced only by free sets, thus once a non free set is found, all its supersets are discarded. FUN selects and validates, step by step, free sets of increasing size. Like TANE [HKPT99], FUN also discovers attributes that are single keys and removes them from the remaining computations to prune the search space.

In [YHB02], the authors propose FD_Mine, a top-down approach inspired by TANE [HKPT99], which discovers FDs by comparing the number of partitions found in the data. To be scalable, FD_Mine combines new prunings with prunings already proposed in TANE. In the first pruning, given two sets of attributes X and Y, if $X \leftrightarrow Y$ is true, then the sets of attributes are considered as equivalent and one of the sets is removed. As a consequence to this pruning all the supersets of the eliminated set will not be checked. Another pruning is based on the *nontrivial closure* of an attribute X, defined as Closure'(X) = $\{Y | X \rightarrow Y\} - \{X\}$. If Closure'($X$) and Closure'($Y$) are the nontrivial closures of attributes X and Y, respectively, then $XY \rightarrow$ Closure'(X) $\cup$ Closure'(Y) should not be checked. This pruning represents the monotonic pruning of FDs. For example, if $X \rightarrow A$ and $Y \rightarrow B$, then $XY \rightarrow A$ and $XY \rightarrow B$ should not be checked. Applying prunings of TANE and new prunings presented above, the authors of FD_Mine show that their approach outperforms both TANE and FUN in the considered datasets.

Top-down approaches discover FDs of increasing size and exploit the monotonic characteristic of discovered FDs to prune the search space. Moreover, attributes that refer

to single keys are discovered and pruned, preventing the discovery of non minimal FDs. Finally, given sets of attributes that are found to be mutually dependent, only one of them is needed in the discovery of FDs. Finally, using a very compact way to represent the values of a relation, this kind of approaches discover the complete set of minimal FDs using a part of the initial data. Using this data representation and different pruning techniques, top-down approaches manage to compute all the FDs without checking all the initial candidate FDs represented in the lattice.

Bottom-up approaches. Bottom-up approaches rely on the fact that to ensure that a FD is valid all the tuples have to be checked, while to discover that a FD is invalid, only two tuples are needed. Thus, these approaches discover first, *maximal invalid dependencies* found in the data and then derive minimal functional dependencies from the invalid ones. An invalid functional dependency $X \rightarrow A$ is maximal when there does not exist $X \subset Z$ for which $Z \rightarrow A$ is invalid.

The set of minimal functional dependencies is called positive cover while the set of maximal invalid dependencies is called negative cover. Given a pair of tuples of a relation, a bottom-up approach returns the set of invalid FDs that are added to negative cover thanks to this pair. This process is done for every pair of tuples in the relation and only the maximal invalid FDs are kept in the negative cover. Once all the maximal invalid FDs are discovered, all the minimal $X \rightarrow A$ that are not subsets of a invalid FD correspond to a minimal FD. Discovering the positive cover using the negative cover has been characterized as the bottleneck of the bottom-up approaches.

For example, in the Table 2.2, the process starts from the tuples t1 and t2. In this couple, the maximal invalid FDs found are $A \rightarrow B$ and $A \rightarrow C$. Continuing with the tuples t1, t3 the maximal invalid FD is $AB \rightarrow C$ while in the t2, t3 the $AC \rightarrow B$. The final negative cover is $AB \rightarrow C$, $AC \rightarrow B$. All the minimal valid FDs are more general than an invalid FD. Therefore, in this example, the positive cover is $\emptyset \rightarrow D$, $B \rightarrow A$, $C \rightarrow A$.

R_1	A	B	C	D
$t1$	1	2	3	5
$t2$	1	3	4	6
$t3$	1	2	4	7

Table 2.2 Tuples of the relation $R_1(A,B,C,D)$

In [SF93], the authors propose a bottom-up approach that discovers first the negative

cover and then uses it to derive the positive cover. To discover the negative cover, [SF93] searches for invalid FDs in every pair of tuples. In this work, the derivation of FDs from the invalid FDs is done in a top-down way. The algorithm for the derivation of the positive cover is composed of two parts. Starting from the most general candidate FDs, the algorithm checks if a candidate FD refers to an invalid FD using the negative cover and if it is found to be invalid, new attributes are added in its left-hand size until a valid FD is found. The process continues until all the valid FDs are derived.

In [FS99], the same authors propose a bottom-up approach based on [SF93] using a more efficient algorithm for deriving the positive cover from the negative cover. Unlike [SF93], this new algorithm is driven by the negative cover and it finds for each discovered invalid FD the valid FDs. Thus, it derives the positive cover from the negative cover in a bottom-up way. Comparing with TANE, [FS99] appears to be more efficient when the number of attributes is big.

In bottom-up approaches, the discovery of FDs is done in two steps, first the complete set of maximal invalid FDs are discovered and second these invalid FDs are used to derive the set of minimal FDs. Bottom-up approaches are based on the fact that an invalid FD can be discovered using only two tuples, unlike valid FDs. Different strategies are proposed to improve the efficiency of FD derivation.

Hybrid approaches. Hybrid approaches combine top-down and bottom-up strategies to discover minimal FDs.

In [LPL00], the authors propose Dep-Miner, an approach that uses a bottom-up algorithm to discover the negative cover and a top-down algorithm to derive the positive cover. As TANE [HKPT99], Dep-Miner represents the data in a stripped partition database in order to have a compact representation of the data. Unlike the previous approaches that compute the negative cover by finding invalid FDs in each pair of tuples of the dataset, Dep-Miner uses the stripped partition database to filter out tuples that do not share values. Indeed, tuples that never share values cannot lead to an invalid FD, thus they are not explored.

To find the minimal FDs of the form $X \rightarrow A$, Dep-Miner selects a right-side part A and using all the invalid FDs having A as right-side part, it adds attributes until all the minimal FDs are found.

The authors of [WGR01] propose an extension of Dep-Miner [LPL00] called FastFDs. Like Dep-Miner FastFDs stores the data in a stripped partition database in order to identify

pairs of tuples that should not be explored. FastFDs discovers directly only the positive cover in a relation. The experiments show that FastFDs can outperform Dep-Miner for the considered relations.

Hybrid approaches take advantage of both bottom-up and top-down approaches to reduce the number of computations needed to find the complete set of minimal FDs.

Approaches discovering valid FDs in several relations. In [WDS$^+$09], the authors propose a way of retrieving non composite probabilistic FDs from a set of sources, based on a mediated schema and schema mappings between the sources. A probabilistic FD denoted by $X \rightarrow^p A$, where p is the likelihood of $X \rightarrow A$ to be a valid FD. To obtain probabilistic FDs for a number of sources, two different strategies are proposed: the first merges data coming from relations of different sources before discovering FDs, while the second discovers FDs in each relation of a source and then merges them. The first strategy is more useful for sources containing relations with few tuples and incomplete information. Nevertheless, this strategy may introduce noise since different representations may be used to describe same real world objects. In sources containing relations with many tuples, the second strategy seems more appropriate.

Conditional functional dependencies discovery in relational databases

Unlike a functional dependency that must hold for all the tuples of a relation, a conditional functional dependency, expresses a functional dependency that is conditionally valid, only in a subpart of the data. Conditional functional dependencies have received a lot of attention the last years, having as main applications to summarize data semantics, to detect and repair data inconsistencies.

Definition 5. *(Conditional functional dependency).* A *conditional functional dependency (CFD)* φ on R is a pair (R: $X \rightarrow A$, t_p) where (1) X is a set of attributes in $attr(R)$ and A a single attribute, (2) $X \rightarrow A$ is a standard FD, referred to as the FD embedded in φ; (3) and t_p is a *pattern tuple* with attributes in X and A where each $B \in X \cup A$, $t_p[B]$ is either a constant $'a'$ in the $dom(B)$ or an unnamed variable $'\_'$ that draws values from $dom(B)$.

For example in the Figure 2.5 some of the valid CFDs are:
φ_0 : $([CC, ZIP] \rightarrow STR, (44, \_||\_))$
φ_1 : $([CC, AC] \rightarrow CT, (01, 908||MH))$
φ_2 : $([CC, AC] \rightarrow STR, (01, 212||EDI))$

The CFD φ_0 states that when two customers are in the country $CC = 44$, the *ZIP* uniquely determines the street *STR*. $(44, \_||\_)$ refers to the pattern tuple that where this FD is valid.

As shown in [GKK+08], given a fixed FD $X \rightarrow A$, the problem of discovering pattern tuples associated to this FD is NP-complete.
A CFD $X \rightarrow A$ is minimal when A is functionally dependent on any subset of X.

Classification of CFDs. The CFDs can be classified in two categories, the constant CFDs and the variable CFDs. A CFD $(X \rightarrow A, t_p)$ is called constant CFD if its pattern tuple t_p consists only of constants, i.e., $t_p[A]$ is a constant and for all $B \in X$, $t_p[B]$ is a constant.
It is called variable CFD if $t_p[A] =' \_'$, i.e., the right part side of its pattern tuple is the unnamed variable $'\_'$. The CFD φ_0 is considered as variable while φ_1 and φ_2 are considered as constant.

To measure the quality of a CFD, the notion of support can be used. The support S_φ of a CFD φ represents the the number of tuples that satisfy the pattern tuple of φ to the CFD to the complete number of tuples in the relation.

A CFD is considered as *frequent* if its support is greater than a given threshold.

The CFDs were initially proposed by [Mah97]. Both [Mah97] and [BFG+07] assume that the CFDs are known and propose SQL based techniques to clean the data using this knowledge. In [GKK+08], the authors state that even if $X \rightarrow Y$ is known to be a CFD, the pattern tuples where this CFD is valid might be unknown. For this reason, the authors of [GKK+08] propose an approach that computes close-to-optimal pattern tuples for a CFD when the FD is given. However, [Mah97, BFG+07, GKK+08] assume that the FDs are available which is not usually the case. Thus, several works focus on the automatic discovery of CFDs.

The problem of CFDs discovery is first addressed in [CM08]. This approach is inspired by TANE [HKPT99] and discovers minimal CFDs using an attribute lattice as seen in Figure 2.4. As in TANE, this approach traverses the lattice in a top-down way starting the exploration from the most general candidate CFDs in order to obtain only minimal CFDs. This work uses a stripped partition database to store a relation in a compact way. Once a candidate CFD $X \rightarrow A$ is generated, to consider if it is valid, at least one equivalent class of X should refine an equivalent class of A. This means a specific value of X determines a specific value of A. Several prunings are applied to make the approach more scalable. As already

seen in FD discovery, when for two attribute sets X and Y it is true that $|\pi_Y(R)| = |\pi_X(R)|$ and $Y = X \cup \{A\}$ where A is one attribute, then $X \rightarrow A$. Thus, to improve the efficiency of CFD discovery, [CM08] searches first if a candidate corresponds to a FD and it filters it out if the above condition is true. Moreover, if a candidate $X \rightarrow A$ is a functional dependency over R, no candidates having as left part supersets of X and as right side A will not be explored. The support of X is also used to avoid the search of some candidates in the lattice. Considering that a set X is not large enough (given a threshold θ), since then the addition of more properties will be equal or smaller, CFDs containing X as a left-hand side are avoided.

The authors of [FGLX11] propose three new algorithms for the discovery of CFDs, CTANE, CFDMiner, and FastCFD, each one efficient in different cases. Like [CM08], CTANE is based on TANE and discovers directly minimal variable CFDs in a top-down way. CTANE extends the prunings of TANE to improve the scalability of CFD discovery. FastCFD is a bottom up approach based on FastFD, an approach proposed in [WGR01] for the discovery of FDs. Unlike the other two algorithms, CFDMiner discovers only constant CFDS. The authors show through their experimental evaluation that top-down approaches such as CTANE may not scale when the number of attributes increases. In addition, CFDMiner has been proved to be the most efficient since, discovering constant CFDs is faster than discovering variable CFDs.

Finally, [LLTY13] proposes a top-down approach for the discovery of constant CFDs applying more pruning techniques to reduce the search space. Unlike any other top-down approach, the authors use a lattice to represent combinations of values. Each level of the lattice corresponds to combinations of values of the same size. More precisely, the first level represents every single value in the data, the second level represents combinations of two values and so on. Additionally, for each combination of values, its support is also stored. Given two nodes X and $Y = X \cup A$, the combination of values X determines the value A (i.e., $X \rightarrow A$) if the $support(X) = support(Y)$. The scalability of this work are based on the pruning of nodes in the lattice. A node is pruned when it and its descendants cannot lead to new minimal CFDs. Thus, the monotonic characteristic of a CFD is exploited. Moreover, this work considers as candidate CFDs sets of values that have a minimum support. Thus, if the support of a combination is less than a given threshold, this combination and all its supersets are eliminated.

Key discovery in relational databases

Keys are usually unknown. There exist approaches that aim to discover keys from the data. In general, these approaches are interested in discovering minimal keys. Since the problem

of key discovery is a subproblem of FD discovery, an approach can compute the set of minimal keys considering that the complete set of FDs is given [SS96]. Nevertheless, the complete set of FDs is rarely known and its discovery costs more than the discovery of keys.

The problem of discovering minimal composite keys (i.e., minimal keys that are composed of more than one attribute) in a given dataset, when no FDs are known, is #P-hard [GKM$^+$03]. Indeed, to check if a set of properties is a key, a naive approach would scan all the tuples of a table to verify if there are no tuples sharing values for these attributes. Even in the case where every key is composed of few attributes, the number of candidate keys can be millions. For example, considering a relation described by 60 attributes the number of candidate keys would be 2^{60} -1. Even if we know that all the existing keys are composed of at most 5 properties, the number of candidate keys is more than six millions ($\binom{60}{1}+\binom{60}{2}+\binom{60}{3}+\binom{60}{4}+\binom{60}{5}$). Thus, several methods to prune the search space have been proposed.

Discovering keys in a relational database has been addressed by various approaches ([SBHR06], [AN11], [VLM12], [KLL13], [HJAQR$^+$13]). These approaches can be organized in two categories, the *row-based approaches* and the *column-based approaches*.

R_2	CC	AC	PN	NM	STR	CT	ZIP
$t1$	01	908	1111111	Rick	Tree Ave.	MH	07974
$t2$	01	908	2222222	Jim	Elm Str.	MH	07974
$t3$	01	212	2222222	Joe	5th Ave	MH	01202
$t4$	44	131	3333333	Ben	High St.	EDI	EH4 1DT
$t5$	44	131	4444444	Ian	High St.	EDI	EH4 1DT
$t6$	44	908	4444444	Ian	Port PI	MH	W1B 1JH

Fig. 2.5 Tuples of the relation $R_2(CC, AC, PN, NM, STR, CT, ZIP)$

Row-based approaches. A row-based approach is based on a row-by-row scan of the database for all column combinations. This strategy is similar to bottom-up approaches where the FDs are derived from the invalid FDs.

The authors of [SBHR06] propose Gordian, a row-based approach that applies several pruning techniques to improve the scalability of the key discovery. Gordian avoids checking all the values of all the candidate keys by discovering first all the sets of attributes that are not keys, and then uses them to derive the keys. The main idea of Gordian is based on the fact that to validate that a set of attributes is not a key, i.e., a *non key*, only two tuples sharing

values on this set of attributes are needed.

The data are represented in a prefix tree where each level of the tree corresponds to an attribute of the relation. Each *node* contains a variable number of *cells*. Each non-leaf cell has a pointer to a child node. Each cell contains a value in the domain of the attribute corresponding to the level of the node node's level and (ii) an integer that represents the number of tuples of the father node sharing this value. For example, Figure 2.6 shows the prefix tree of the first four attributes CC, AC, PN, NM of the relation $R(CC, AC, PN, NM, STR, CT, ZIP)$ presented in Figure 2.5. The cell 01 in the first level represents three people that share this value for the attribute CC. From these three people, two of them share the value 908 for the attribute AC while one of them has the value 212. Observing the prefix tree, it is easy to deduce that the set of attributes $\{CC, AC\}$ is a non key since two people share values for CC and AC.

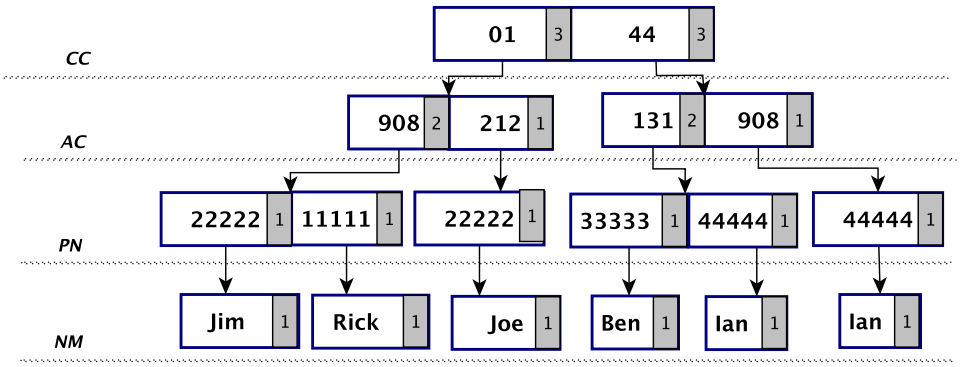


Fig. 2.6 A prefix tree of the attributes CC, AC, PN and NM in relation R_2 of the Figure 2.5

To discover the complete set of maximal non keys, Gordian explores the prefix tree in a depth-first way. Starting from a cell of the root node of the tree, Gordian continues with cells that are in children nodes of the considered cell. When a cell of a leaf node is reached, Gordian discovers that a selected set of attributes, corresponding to the traversed cells, refers to a non key if there exist more than two objects sharing values for this set of attributes. To compute all the possible combinations of attributes, Gordian "forgets" levels of the prefix tree by merging the children nodes of the following levels and discovers non keys on subparts of the prefix tree. For example, to discover the non key $\{CC, PN, NM\}$, Gordian suppresses the level containing the attribute AC and merges the nodes of the attribute PN. Figure 2.7 shows the new prefix tree.

To optimize the prefix tree exploration, Gordian applies different prunings. First, Gor-

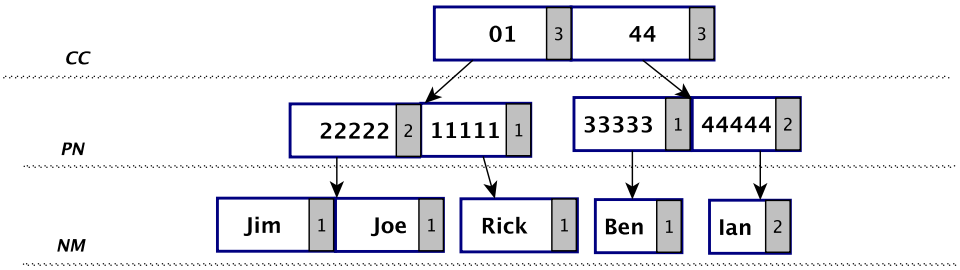


Fig. 2.7 A prefix tree of the attributes *CC*, *PN* and *NM*

dian exploits the anti-monotonic characteristic of a non key by avoiding exploring subparts of already discovered non keys. Second, paths of the prefix tree describing values of only one tuple are avoided since no non key can be discovered. This pruning exploits the monotonic characteristic of a key.

Once the complete set of maximal non keys is discovered, to derive the set of minimal keys from the set of maximal non keys, Gordian computes the Cartesian product of the complement sets of maximal non keys. This derivation is considered as the bottleneck of the approach when first the dataset is composed of many attributes and second the number of maximal non keys is large.

The authors state that Gordian is much faster than a simple column-based approach in all the experiments that have been done.

R_1	A	B	C	D
$t1$	1	2	3	5
$t2$	1	3	null	6
$t3$	1	2	4	7

Table 2.3 Tuples of the relation $R_1(A,B,C,D)$ containing a null value

In [KLL13], the authors introduce the notion of keys under the presence of null values. In Table 2.3, one value of the tuple $t2$ in relation R_1 has been replaced by a null value. Two new types of keys are introduced to represent the effects of the null values on the attributes composing a key, the *possible key* and the *certain key*. A key is considered as possible when the replacement of a null value by a real value can turn this set of properties to a non key, depending on the value. For example, if the null value of the Table 2.3 is replaced by the value 3 or 4, then the attribute C is a non key while if it is replaces by any other value, it is a key. A set of properties that is not affected by the replacement of null values with any value

is called certain key. For example, the set of attribute $\{B,C\}$ is a certain key since whatever value is given to the null value, there do not exist two tuples sharing values for this set.

In the same way, the authors define the notion of *weak non key* and *strong non key*. Given two tuples t, t', a weak non key refers to a set of attributes where for each attribute A, $t[A] = t'[A]$ or $t[A] = null$ or $t'[A] = null$. For each attribute A in a strong non key, $t[A] = t'[A] \neq null$.

To discover the set of certain keys and possible keys, the authors propose a simple row-based approach to discover the maximal weak non keys and strong non keys by exploring all the pairs of tuples of a dataset. In our example, in the tuples $t1$, $t2$, the maximal weak non key is the set $\{A,C\}$ and the strong non key is the attribute A. In the tuples $t1$, $t3$, both the maximal weak non key and maximal strong non key are the set $\{A,B\}$. Finally, for the tuples $t2$, $t3$, the maximal weak non key is $\{A, C\}$ and the maximal strong non key is $\{A\}$. In the end, the set of maximal weak non keys is $\{A,B\},\{A,C\}$ and the set of strong non keys is $\{A,B\}$. Given the set of maximal (weak) strong non keys the set of (certain) possible keys are derived using a key derivation method similar to the one used by Gordian [SBHR06]. For example, given the set of maximal weak non keys $\{A,B\},\{A,C\}$, the Cartesian product of the complement sets $\{C,D\}$ and $\{B,D\}$ gives the sets $\{B,C\}$ and $\{D\}$ which refer to the certain minimal keys in R_1.

Column-based approaches. A column-based approach searches for keys column-by-column. More precisely, the idea is to check the uniqueness of every column combination on all rows. This strategy is similar to top-down approaches used in FDs.

The column-based approach HCA proposed in [AN11] that discovers the set of minimal keys. HCA uses several prunings to reduce the search space. It exploits first functional dependencies found during the key discovery process, to deduce new keys without having to validate them. For example, if the set of attributes $\{A,B\}$ is a key and $C \rightarrow A$, then the set of attributes $\{B,C\}$ is also a key. Moreover, HCA uses partitions of combinations of attributes to prune the search path. For example, in the relation R_2 of the Figure 2.5, the set of attributes $\{CC,CT\}$ cannot be a key since $|\pi_{CC}(R_2)| \times |\pi_{CT}(R_2)| = 4 <$ # tuples. The authors propose also a fusion of their initial approach HCA with [SBHR06] called HCA-Gordian. This new approach runs Gordian in a sample of data and uses the discovered non keys to prune the search of HCA.

Hybrid approaches. To scale in big datasets, hybrid approaches use both column-based and row-based pruning techniques to discover both keys and non keys at the same time.

In [HJAQR+13], the authors propose DUCC, a hybrid approach for the discovery of minimal keys. As in TANE [HKPT99], all the possible combinations of attributes are represented in a lattice. A preprocessing step is applied to filter out all the single keys. DUCC exploits the monotonic characteristic of keys and the anti-monotonic characteristic of non keys to prune the search space of the lattice. This means that when a key (non key) is discovered, no superset (subset) of this key will be explored since, by definition, it refers to a key (non key). Pruning the lattice using keys and non keys might create unreachable nodes in the lattice. To discover these nodes, DUCC checks if the set of discovered non keys leads to the set of discovered keys, using the key derivation algorithm proposed by Gordian [SBHR06]. Finally, to improve the efficiency of the approach, DUCC uses parallelization techniques to test different sets of attributes simultaneously.

In [VLM12], the authors propose two different approaches for discovering keys on a sample of data provided by a domain expert. The first proposition is a row-based approach that (i) discovers maximal non keys for a set of tuples and (ii) adds attributes to the sets of non keys until they become keys. Only minimal keys are kept.

Their second proposition is an approach that discover non keys using a row-based way to discover keys and a column-based way to derive keys from non keys. Similar to the dep-Miner approach [LPL00], this approach discovers first the set of maximal non keys using a row-based algorithm and then derives the set of minimal keys using a column-based algorithm.

2.2.2.2 Discovering keys or semantic dependencies in Semantic Web

The problem of key discovery in RDF datasets, in the setting of the Semantic Web, is similar to the key discovery problem in relational databases. Nevertheless, in database area the approaches do not consider the semantics defined in the ontology. Unlike RDF data, the properties of a relational table cannot be, by construction, multivalued. Moreover, RDF data may be incomplete and may contain duplicates or erroneous data. For these reasons, different approaches are needed to discover keys in RDF datasets. In this section, we present first the few approaches that have faced the keys discovery in RDF data. Before that, we present an approach that discovers semantic dependencies, a problem that is similar to the key discovery problem.

Semantic dependency discovery

Similar to FDs and CFDs in relational databases, semantic dependencies represent knowledge of the data that is more general than keys. Considering that the data published on the Web are usually incomplete, extracting semantic dependencies from the data can help to make them more complete. Moreover, potential errors in an RDF dataset might be detected using semantic dependencies. Finally, they can be also exploited by approaches that do reasoning. Since this knowledge is usually unknown, the authors of [GTHS13] propose an approach for their automatic discovery.

Definition 6. *(Semantic dependency).* A *semantic dependency (SD)* over an RDF dataset D is $B_1 \wedge B_2 \wedge \ldots \wedge B_n \Rightarrow p(x,y)$ where B_i corresponds to $p_i(x_i,y_i)$ where each x_i and y_i can correspond to variables or constants. $B_1 \wedge B_2 \wedge \ldots \wedge B_n$ corresponds to the body of the SD while $p(x,y)$ to the head of the SD.

For example, the SD $s_1 : motherOf(m,c) \wedge marriedTo(m,f) \Rightarrow fatherOf(f,c)$ states that if m is the mother of c and she is married to f, then f is the father of c.

The authors of [GTHS13] propose AMIE, a top-down approach that discovers semantic dependencies in RDF data. To be capable to discover SDs under the OWA, the authors assume that if at least one property value is given for an instance, then all the property values for this instance are given (local completeness). One of the main problems of such approaches is to find an efficient way to discover all the possible SDs. To restrict the search space, AMIE discovers only *closed connected semantic dependencies*. An SD is closed if every variable in the SD appears at least twice and connected if every B_i is transitively connected to every other atom of the rule. For example, the SD s_1 seen before is both closed and connected. AMIE starts from the most general candidate SDs, and it adds B_i until a candidate SD becomes valid. Once a SD is discovered, more specific SDs are not explored. To improve the execution time of the discovery, different candidates are tested in parallel. Moreover, the authors use the measure of *head coverage* to reduce the search space. This measure is computed using the *support* of a SD which represents the number of pairs x and y for which the SD $B_1 \wedge B_2 \wedge \ldots \wedge B_n \Rightarrow p(x,y)$ is valid. The head coverage is:

$$hc(B_1 \wedge B_2 \wedge \ldots \wedge B_n \Rightarrow p(x,y)) = \frac{support(B_1 \wedge B_2 \wedge \ldots \wedge B_n \Rightarrow p(x,y))}{\#(x',y') : p(x',y')}$$

The authors are only interested in SDs having head coverage bigger than a given threshold. In this way, SDs covering only few cases are not computed.

Since there can exist SDs with few exceptions, the authors propose the *PCA confidence* measure to compute the confidence of a SD in the setting of OWA. The PCA confidence is computed as follows:

$$pcaconf(B_1 \wedge B_2 \wedge \ldots \wedge B_n \Rightarrow p(x,y)) = \frac{support(B_1 \wedge B_2 \wedge \ldots \wedge B_n \Rightarrow p(x,y))}{\#(x,y) : \exists z_1, \ldots, z_m, y' : B_1 \wedge B_2 \wedge \ldots \wedge B_n \wedge p(x,y')}$$

Key discovery in RDF data

Keys in RDF data are not usually specified. Discovering automatically keys from the data is #P-hard [GKM$^+$03]. The problem of key discovery in RDF data has been addressed by few works. The presented approaches discover different kinds of keys. To evaluate the quality of keys, different measures are proposed.

In PARIS [SAS11], the authors discover (inverse) functional properties from datasets where the UNA is fulfilled. An (inverse) functional property is a single key applicable to all the instances of the data. In this work, one ontology contains both the schema and the instances. The aim is to link instances and schema entities (i.e.,classes and properties). To link instances, PARIS discovers (inverse) functional properties in the datasets where the UNA is fulfilled.

To avoid the loss of (inverse) functional properties due to erroneous data, the authors compute the (inverse) functionality degree which represents the probability of a property to be (inverse) functional. This measure is also used to capture properties that are not (inverse) functional due to few exceptions. The functionality degree of one property is:

$$fun(p) = \frac{\#x \quad \exists y : p(x,y)}{\#x,y \quad p(x,y)}$$

The inverse functionality degree is defined analogously. The (inverse) functionality degree of a property is low when there exist many distinct (instances) values referring to the same (values) instances. Thus, only properties with high (inverse) functionality degree are useful.

As seen in the Section 2.2.1.1, [SH11] is a blocking approach that discovers discriminative properties. In this work, only datatype properties are considered. The discriminability is similar to the inverse functionality degree defined in [SAS11]. The only difference is that the discriminability is defined for instances that belong to a set of given classes. Indeed, a property can be highly discriminative for a class and not for another. For example, the name is a very discriminative property for a country but not for a person.

The discriminability is:

$$dis(p, I_C) = \frac{\#y \;\; p(x,y) \wedge x \in I_C}{\#x,y \;\; p(x,y) \wedge x \in I_C}$$

where I_C represents the set of instances of every selected class $c \in C$. The authors propose also a new metric, to measure the proportion of instances where a property is instantiated among the total number of instances. This measure is called *coverage* of a property p. A low coverage represents a property that is not used to describe many instances, thus is not useful.

The coverage is:

$$cov(p, I_C) = \frac{\#x \;\; p(x,y) \wedge x \in I_C}{|I_C|}$$

For example, even if the discriminability of a property is 1, i.e., there exists only unique values for this property, if we consider that there exist 50 instances for the set of classes C and that this property appears only in 5 instances (coverage=5/50 = 0.1), then the authors do not consider this property as significant.

To take into account both metrics, the authors introduce the notion of $F1$-*score* (F_L):

$$F_L = \frac{2 * dis(p, I_C) * cov(key, I_C)}{dis(p, I_C) + cov(p, I_C)}$$

F_L is computed for every property p. Given a threshold α, if there exist several properties with $F_L > \alpha$, only the property with the highest F_L is selected. If none of the properties satisfy this condition, the algorithm combines the property that has the highest discriminability with every other property in the dataset. The F_L of the new combinations of properties is computed and if no combination satisfies the condition, the process continues until one combination of properties has F_L higher than the threshold. Thus, the number of computations depends on the defined threshold.

This approach discovers only a subpart of the keys that can be found in the data.

In [ADS12], the authors present an approach that discovers keys and pseudo-keys (keys that hold in a part of the data) for data linking and data cleaning. The keys that are discovered in this work use a definition, different from the one defined in OWL2. Considering a property p and an instance i, $p(i)$ denotes the set of values that the instance i has for the

property p. In this approach, a key in one dataset is defined as follows:

Definition 7. *(Key).* A set of properties P is a key if for all the instances i_1, i_2, if $p(i_1) = p(i_2)$ for all $p \in P$ then $i_1 = i_2$.

This type of keys is mainly significant for datasets where the data are *locally complete*, i.e., for each instance and for each property, all the values are considered to be given.

For example, using the extract of an RDF dataset of the Figure 2.8, the property *FirstName* is a key since, the sets of values for this property $\{Joan, Armando\}$, $\{Joan\}$ and $\{George\}$ are distinct. Unlike this approach, the property *FirstName* would not be considered as a key using the semantics of OWL2 since there exist two different people that have the same name.

$FirstName(p1, "Joan"), FirstName(p1, "Armando"), LastName(p1, "Robles"),$
$FirstName(p2, "Joan"), LastName(p2, "Galarraga"), FirstName(p3, "Alex"),$
$LastName(p3, "Galarraga"), owl{:}different(p1, p2), owl{:}different(p1, p3),$
$owl{:}different(p2, p3)$

Fig. 2.8 Extract of an RDF dataset

The discovery of keys is done using a bottom-up approach based on TANE [HKPT99]. Similarly to TANE, the data are stored in a structure similar to a stripped partition database with the only difference that partitions of size one are also kept. To consider a set of properties as a key, each partition should have size one (i.e., singleton partition). To validate the importance of a discovered key, the authors use the *support* which is the same as the coverage proposed by [SH11].

In order to discover sets of properties that are either keys under the presence of erroneous data or not keys due to few exceptions, the authors introduce a new type of keys called *pseudo-keys*. To measure the quality of such sets of properties a new notion of *discriminability* different from the one proposed in [SH11] is given.

For a set of properties P, the discriminability is:

$$dis(P) = \frac{\#\ singleton\ partitions}{\#\ partitions}$$

A set of properties P is considered as pseudo-key when the discriminability of P is bigger than a given threshold.

For example, if a set of properties is considered as pseudo-key if the discriminability is bigger than 0.3 then the property *LastName*, in the Figure 2.8, will be a pseudo key ($dis(LastName) = 0.33$).

2.3 Discussion

Identity links between resources of different datasets add a real value to the linked data existing on the Web. Due to the huge amount of available data and to its heterogeneity, it is difficult to set the identity links manually. Therefore, there exist many approaches that focus on the automatic discovery of identity links.

Some data linking approaches are numerical and use complex similarity measures, aggregation functions and thresholds to build the rules. These approaches are often adapted to a given pair of datasets. This kind of rules in some cases is automatically discovered [IB12, NL12, NdM12, SAS11]. Other approaches are based on logical rules [SPR09, HCQ11] that are automatically generated using the semantics of keys or (inverse) functional properties. This kind of knowledge can also be used by an expert to construct more complex similarity functions that take more into account properties that are involved in keys [VBGK09, NA11]. Moreover, for scalability reasons, keys can be involved in blocking methods to create blocks of instances that are possibly referring to the same real world object [SH11].

The advantage of using keys instead of complex similarity functions is that keys can be valid to a given domain and not only to a specific pair of datasets. Since it can be hard even for an expert to define the complete set of keys and such knowledge is not usually declared in an ontology, approaches that automatically discover keys from the data are needed.

The problem of key discovery has been previously studied in the setting of relational databases. The key discovery approaches in relational databases are based on strategies initially proposed for the discovery of FDs. Since the discovery of keys is an #P-hard problem, different strategies to reduce the size of the search space are proposed. Key discovery approaches apply pruning strategies such as the monotonicity of keys and the anti-monotonicity of non keys that are inspired from FD discovery approaches. Moreover, structures for storing the data in an efficient way like the stripped partition database or a prefix tree are proposed to represent the data in a compact way. The key discovery approaches can be grouped into two main categories, the column-based approaches and the row-based approaches.

Column-based approaches, based on top-down approaches for FDs, focus on the dis-

covery of the complete set of minimal keys. These approaches build candidate keys and validate them against the data. Already discovered keys are used to prune the search space (i.e., monotonic pruning). Unlike column-based approaches, row-based approaches, based on bottom-up approaches for FDs, search the data to discover first all the maximal non keys and then derive the minimal keys from the discovered non keys. The pruning of the search space is based on the anti-monotonic characteristic of non keys. These approaches seem to be more efficient than column-based approaches when many data should be explored and the number of properties is big. Intuitively, discovering a non key from the data is much more easy than discovering a key. The bottleneck of such approaches is the derivation of minimal keys from the set of maximal non keys.

Even if the key discovery problem in the setting of relational databases is similar to the one in the setting of Semantic Web, different characteristics of the RDF data should be taken into account. Since RDF data are usually composed of many triples and properties, new pruning strategies and structures for storing efficiently the data, taking into account the semantics of the ontology, are required. Unlike RDF data that contain multivalued properties, approaches that discover keys in the setting of relational databases find keys that are valid only in one relational table. Moreover, RDF data usually conform to an ontology where knowledge such as hierarchies of classes exist. Very often, RDF datasets contain incomplete data. To deal with this, different assumptions for the not declared information in the data should be considered. Most of the approaches proposed both in relational databases and the Semantic Web consider that the data used in the discovery are locally complete. Only the approach [KLL13], introduced in the setting of relational databases, proposes different heuristics of interpreting the null values in a relational table.

The problem of discovering the complete set of minimal composite keys in a given RDF dataset has been addressed only in [ADS12]. Nevertheless, in this work, the authors provide an approach that discovers keys that do not conform to the semantics of keys as it is provided by OWL2 (see [ACC+14] for a comparative study). This work does not provide pruning strategies that can improve the scalability of the discovery. The remaining approaches focus only on the discovery of either single keys or a small subset of composite keys. All these approaches discover keys considering that all the data are locally complete.

To estimate the quality of the discovered keys, different metrics such as the support and the discriminability of each key are used. None of the existing approaches propose a strategy for merging keys discovered in different datasets. A merge strategy may allow the discovery of keys with higher quality since keys globally valid may be more significant than keys locally discovered in different datasets.

Thus, we consider that approaches that discover efficiently the complete set of minimal OWL2 keys, taking into account erroneous data duplicates, are needed. Since RDF data can be incomplete, different heuristics are required. Also, strategies for merging keys discovered in different datasets are necessary. To the best of our knowledge, there do not exist approaches that discover *conditional keys*. A conditional key is a set of properties that is a key for a subpart of the data. Similar to keys and FDs, conditional keys are similar to CFDs. CFD approaches exploit strategies first proposed in the FD discovery to discover the complete set of CFDs. Since conditional keys can be used to enrich the knowledge for a given domain, their discovery can be interesting.

To summarize the contributions of all the key discovery approaches that have been studied in this thesis, we provide the comparison Table 2.4.

Semantic Web						
Approach	Composite keys	Complete set of keys	OWL2 keys	Approximate keys	Incomplete data heuristics	Strategy
[SAS11]			✓	✓		-
[SH11]	✓		✓	✓		Column-based
[ADS12]	✓	✓		✓		Column-based
Relational databases						
	Composite keys	Complete set of keys	Approximate keys		Incomplete data heuristics	Strategy
[SBHR06]	✓	✓				Row-based
[AN11]	✓	✓				Column-based or Hybrid
[VLM12]	✓					Row-based
[HJAQR+13]	✓	✓				Hybrid
[KLL13]	✓	✓			✓	Row-based

Table 2.4 Comparison of key discovery approaches

Chapter 3

KD2R: A Key Discovery approach for Data Linking

In the recent years, keys play an increasingly important role in the Semantic Web, especially thanks to the introduction of *owl:HasHey* construct in OWL2. This construct allows to declare an axiom in an ontology stating that a set of properties is a key for a specific class. Keys can be used for different purposes. They can be exploited as logical rules to deduce identity links with a high precision rate. Moreover, keys can guide the construction of more complex linking rules, including elementary similarity measures or aggregation functions specified by user experts. Similarly to blocking methods for relational data, keys can also help to detect instance pairs that do not need to be compared.

In most of the datasets published on the Web, keys are not available. Furthermore, when ontologies are large, composed of many classes and properties, it can be very difficult even for a human expert to determine keys. Single keys that are usually more easy to define, such as the *Social Security Number (SSN)* for a person or the *International Standard Book Number (ISBN)* for a book, are very rare. Thus, an expert might not be able to give the complete set of complete keys and might even provide erroneous ones. Therefore, automatic methods that discover composite keys from the data are necessary.

The problem of automatic key discovery has been already studied in the relational database field. Even if the key discovery problem appears to be similar in RDF data, there exist characteristics of the RDF data that should be taken into account. To begin with, since the discovery of keys in the relational database field always concerns data found on a single relational table, properties cannot be, by construction, multivalued. On the contrary, multi-valued properties are very common in RDF data. Moreover, considering the information of an RDF dataset as complete or not, i.e., Closed World Assumption (CWA) or Open World

Assumption (OWA) should be studied. Finally, issues such as class inheritance exist only in RDF. Thus, the need for tools that handle these RDF characteristics is obvious.

Since the quality of the keys depends on the data, it is evident that the more data are exploited in key discovery, the more probable it is to obtain meaningful keys. Moreover, we note that most of the datasets published on the Web contain classes described by a big number of properties. This makes the discovery of keys even harder. Therefore, methods that can deal with big data and discover keys efficiently are required.

In this chapter, we present KD2R, an automatic approach for composite key discovery in RDF datasets that conform to OWL ontologies. To discover keys in datasets where the CWA cannot been ensured, we theoritically need all the *owl:sameAs* and *owl:differentFrom* links existing in a dataset. Since usually RDF datasets do not contain these links and the CWA cannot be ensured, KD2R discovers keys in datasets where the Unique Name Assumption (UNA) is fulfilled, i.e., there exists an implicit *owl:differentFrom* link for every pair of instances in the data. Moreover, discovering keys when data might be incomplete is also possible. KD2R uses either a pessimistic or an optimistic heuristic to interpret the absence of information in the data. To be more efficient, KD2R discovers first maximal *non keys* (i.e., a set of properties having the same values for several distinct instances) before inferring the keys. Furthermore, thanks to the addition of the ontology, KD2R exploits key inheritance between classes in order to prune the non key search space. In order to obtain keys that are valid in different datasets, KD2R discovers first keys in each dataset and then applies a merge operation to compute them. To find keys in datasets conforming to distinct ontologies, ontology alignment tools are used that create mappings between ontology elements (see [PJ13] for a recent survey on ontology alignment). These mappings are exploited to find keys that are valid in all the datasets.

The work described in this chapter has led to an early publication in the [SPS11]. The final version of this work, which the present Chapter closely follows, appeared in [PSS13]. In [ACC+14, CCL+14], a theoretical and experimental comparison of KD2R with the key discovery tool provided by [ADS12] is conducted.

In the following of this chapter, we present all the contributions of KD2R. The main contributions are:

1. An algorithm for the efficient discovery of OWL2 keys from non keys applying different pruning strategies.

2. The introduction of a pessimistic heuristic and an optimistic heuristic to interpret the

absence of information in incomplete data.

The remainder of this chapter is organized as follows. Section 3.1 provides the problem statement of key discovery. Section 3.2 presents the general idea of KD2R while Section 3.3 gives the key discovery algorithms. In Section 3.4, we present the optimistic heuristic applied in the key discovery. The results of the experiments are described in Section 3.5. We provide our concluding remarks in Section 3.6.

3.1 Problem Statement

OWL2 keys express sets of properties that uniquely identify each instance of a class. However, if a complete knowledge of *owl:sameAs* and *owl:differentFrom* is not provided in a dataset, it is not possible to distinguish the two following cases:

1. common property values describing instances that refer to the same real world entity
2. common property values describing instances that refer to two distinct real world entities.

Dataset *D1*:
Person(p1), $firstName(p1, "Wendy")$, $lastName(p1, "Johnson")$, $hasFriend(p1, p2)$, $hasFriend(p1, p3)$, $bornIn(p1, "USA")$,

Person(p2), $firstName(p2, "Wendy")$, $lastName(p2, "Miller")$, $bornIn(p2, "UK")$,

Person(p3), $firstName(p3, "Madalina")$, $lastName(p3, "David")$, $hasFriend(p3, p2)$, $hasFriend(p3, p4)$,

Person(p4), $firstName(p4, "Jane")$, $lastName(p4, "Clark")$, $bornIn(p4, "Ireland")$

Fig. 3.1 RDF dataset $D1$

The Figure 3.1 presents the dataset $D1$ that contains instances of the class *Person*. A person is described by its first name, its last name, its friends and the country where he/she was born. In $D1$, if we know that the persons $p1$ and $p2$ are the same, the property $firstName$ can be considered as a key.

In the RDF datasets that are available on the Web, *owl:sameAs* and *owl:differentFrom* links are rarely declared. If we consider the overall Linked Open Data cloud (LOD), there

exist datasets containing duplicate instances. Nevertheless, datasets that fulfill the UNA, i.e., all the instances of a dataset are distinct, are not so uncommon. Indeed, datasets generated from relational databases are in most of the cases under the UNA. Furthermore, in some cases RDF datasets are created in a way to avoid duplicates, like the YAGO knowledge base [SKW07]. Thus, we are interested in discovering keys in datasets where the UNA is fulfilled. For example, considering that the dataset $D1$ of the Figure 3.1 is under the UNA, the property $firstName$ cannot be a key since there exist two distinct persons having the same name.

When literals are heterogeneously described, the key discovery problem becomes much more complex. Indeed, syntactic variations or errors in literal values may lead to miss keys or to discover erroneous ones. For example, in the dataset $D1$, if a person is born in *"USA"* and another in *"United States of America"*, *bornIn* can be found as a key. In this work, we assume that the data described in one dataset are either homogeneous or have been normalized.

Furthermore, in the Semantic Web context, RDF data may be incomplete and asserting the Closed World Assumption (CWA), i.e., what is not currently known to be true is false, may not be meaningful. For example, the fact that in the dataset $D1$, the person $p2$ has no friends does not mean that $hasFriend(p2,\ p1)$ is false. Axioms such as functionality or maximum cardinality of properties could be taken into account to exploit the completeness of some properties. Since these axioms are rarely given, discovering keys in RDF data requires the use of heuristics to interpret the possible absence of information. We consider two different heuristics, the optimistic heuristic and the pessimistic heuristic:

- Pessimistic heuristic: when a property is not instantiated, all the values in the dataset are possible while in the case of instantiated properties, we consider that the information is complete. For example, $hasFriend(p2,\ p1)$, $hasFriend(p2,\ p3)$, $hasFriend(p2,\ p4)$ are possible while $hasFriend(p1, p4)$ is not.

- Optimistic heuristic: only the values that are declared in the dataset are taken into account in the discovery of keys. In other words, we consider that if there exist other values that are not contained in the dataset, they are different from all the existing ones. For example, $hasFriend(p2,\ p3)$ is not possible.

The quality of the discovered keys improves when numerous data coming from different datasets are exploited. Thus, we are interested in discovering keys that are valid in several datasets. The datasets may not be described using the same ontology. Hence, we assume that equivalence mappings between classes and properties are declared or computed by an

ontology alignment tool. However, we do not consider that all the datasets are united in a single dataset (under an integrated ontology). Indeed, in this case the UNA would no longer be guaranteed. Therefore, keys are first discovered in each dataset and then merged according to the given mapping set.

Let $D1$ and $D2$ be two RDF datasets that conform to two OWL ontologies $O1$, $O2$ respectively. We assume that OWL entailment rules [PSHH04] are applied on both RDF datasets. We consider in each dataset Di the set of instantiated properties $\mathscr{P}_i = \{p_{i1}, p_{i2}, \ldots, p_{iN}\}$. To discover keys that involve property expressions, we assume that for each object property p, an inverse property (inv-p) is created. Let $C_i = \{c_{i1}, c_{i2}, \ldots, c_{iL}\}$ be a set of classes of the ontology Oi. Let $\mathscr{M}$ be the set of equivalence mappings between the elements (properties or classes) of the ontologies $O1$ and $O2$. Let $\mathscr{P}_{1c}$ (resp. $\mathscr{P}_{2c}$) be the set of properties of $\mathscr{P}_1$ (resp. of $\mathscr{P}_2$) such that there exists an equivalence mapping with a property of $\mathscr{P}_2$ (resp. of $\mathscr{P}_1$). The problem of key discovery that we address in this chapter is defined as follows:

1. for each dataset Di and each class $c_{ij} \in C_i$ of the ontology Oi, such that it exists a mapping between a class c_{ij} and a class c_{ks} of the other ontology Ok, discover the parts of $\mathscr{P}_i$ that are keys in the dataset Di

2. find all the parts of $\mathscr{P}_{ic}$ that are keys for equivalent classes in the two datasets $D1$ and $D2$ with respect to the property mappings in $\mathscr{M}$.

3.2 KD2R: Key Discovery approach for Data Linking

In this section, we introduce some preliminary definitions before presenting an overview of KD2R approach. In the following, we consider that the dataset $D1$ of the Figure 3.1 is under the UNA.

3.2.1 Keys, Non Keys and Undetermined Keys

We consider a set of properties as a key for a class, if every instance of the class is uniquely identified by this set of properties. In other words, a set of properties is a key for a class if, for all pairs of distinct instances of this class, there exists at least one property in this set for which all the values are distinct.

Definition 7. (Key). *A set of properties P ($P \subseteq \mathscr{P}$) is a key for the class c ($c \in \mathscr{C}$) in a dataset D if:*

$$\forall X \, \forall Y \, ((X \neq Y) \wedge c(X) \wedge c(Y)) \Rightarrow$$

$$\exists p_j \ (\exists U \ \exists V \ p_j(X,U) \wedge p_j(Y,V)) \wedge (\forall Z \ \neg(p_j(X,Z) \wedge p_j(Y,Z)))$$

For example in $D1$ (see Figure 3.1), the property *lastName* is a key for the class *Person* since every last name in the dataset is unique. The set of properties $\{firstName, bornIn\}$ is also a key since (i) there do not exist two persons sharing values for the properties $firstName$ and *bornIn* and (ii) whatever is the country that the person $p3$ is born in, this set of properties will always be considered as a key.

We denote $K_{D.c}$ the set of keys of the class c w.r.t the dataset D.

Definition 8. (Minimal key). *A set of properties P is a minimal key for the class c ($c \in \mathscr{C}$) and a dataset D if P is a key and $\nexists P'$ a key s.t. $P' \subset P$*

We consider a set of properties as a non key for a class c if there exist at least two distinct instances of this class that share values for all the properties of this set.

Definition 9. (Non key). *A set of properties P ($P \subseteq \mathscr{P}$) is a non key for the class c ($c \in \mathscr{C}$) and a dataset D if:*

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

For example in $D1$, the property $firstName$ is a non key for the class *Person* since there exist two people having as first name the name *"Wendy"*.

We denote $NK_{D.c}$ the set of non keys of the class c w.r.t the dataset D.

Definition 10. (Maximal non key). *A set of properties P is a maximal non key for the class c ($c \in \mathscr{C}$) and a dataset D if P is a non key and $\nexists P'$ a non key s.t. $P \subset P'$*

To be able to apply the pessimistic and optimistic heuristics, some combinations of properties cannot be considered neither as keys nor as non keys. More precisely, a set of properties is called an *undetermined key* for a class c if (i) this set of properties is not a non key and (ii) there exist at least two instances of the class that share values for a subset of the undetermined key and (iii) the remaining properties are not instantiated for at least one of the two instances.

Definition 11. (Undetermined key). *A set of properties P ($P \subseteq \mathscr{P}$) is an undetermined key for the class c ($c \in \mathscr{C}$) in D if:*

- *(i)* $P \notin NK_{D.c}$ *and*
- *(ii)* $\exists X\ \exists Y\ (c(X) \wedge c(Y) \wedge (X \neq Y) \wedge \forall p_j$
  $((\exists Z\ (p_j(X,Z) \wedge p_j(Y,Z)) \vee \nexists W\ (p_j(X,W) \vee \nexists W\ p_j(Y,W))))$

For example in $D1$, the persons $p1$, $p2$ have the same first name, (*"Wendy"*), but for person $p2$ no information about her friends is given. Thus, the set of properties $\{firstName, hasFriend\}$ is an undetermined key. If we consider that $hasFriend(p2, p3)$ is true in the dataset $D1$, then $\{firstName, hasFriend\}$ is a non key.

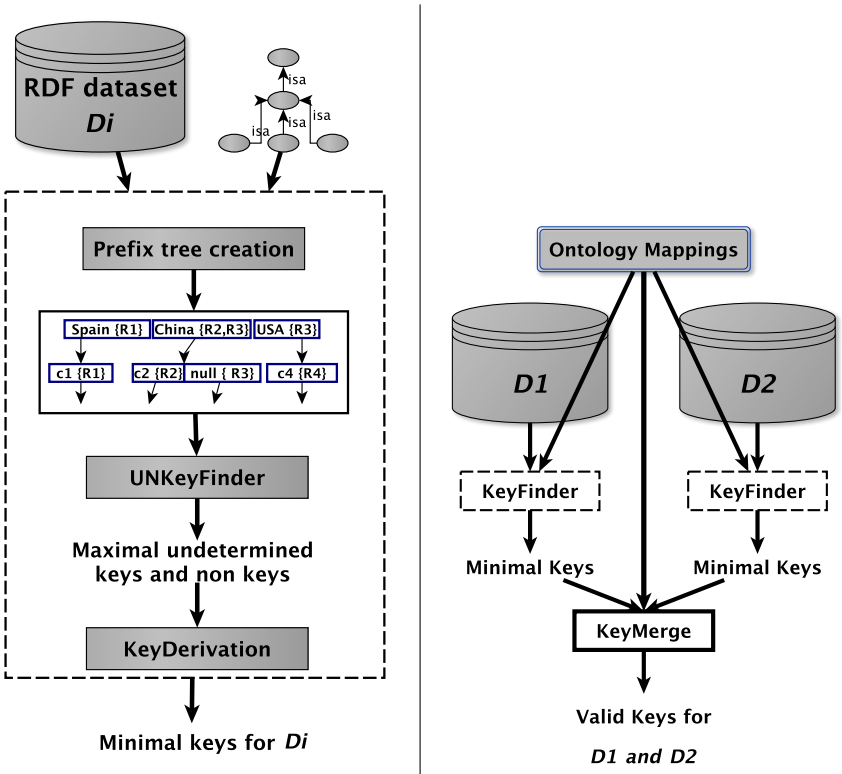
We denote $UK_{D.c}$ the set of undetermined keys of the class c for a dataset D.

Following the Definition 10, an undetermined key P is maximal if there does not exist an undetermined key P' such that $P \subset P'$.

Undetermined keys can be considered either as keys or as non keys, depending on the selected heuristic. Using the pessimistic heuristic, undetermined keys are considered as non keys, while using the optimistic heuristic, they are considered as keys. The discovered undetermined keys can be validated by a human expert who can assign them to the set of keys or non keys.

3.2.2 KD2R overview

A naive automatic way to discover the complete set of keys in a class, is to check all the possible combinations of properties that refer to this class. Let us consider a class described by 60 properties. In this case, the number of candidate keys is $2^{60} - 1$. Even if we consider that the size of each key will be small in terms of number of properties, the number of candidate keys can be millions. In the previous example, if we consider that the maximum number of properties for a key is 5, the number of candidate keys is more than six million. For each candidate key, to ensure if it refers to a key or not, the values of all the instances concerning this candidate key should be explored. In order to minimize the number of computations, we propose a method inspired by [SBHR06] which first retrieves the set of maximal non keys (i.e., sets of properties that share the same values for at least two instances) and then derives the set of minimal keys from the non keys. Unlike keys, having



(a) KeyFinder for one dataset (b) Key merge for two datasets

Fig. 3.2 Key Discovery for two datasets

only two instances sharing values for a set of properties are enough to consider this set as a non key.

In Figure 3.2, we show the main steps of the KD2R approach. Our method discovers the keys for each RDF dataset independently. In each dataset, KD2R is applied on the classes that are previously sorted in a topological order. In this way, the keys that are discovered in the superclasses can be exploited when keys are discovered in their subclasses. For a given dataset *Di* and a given class *c*, we apply KeyFinder (see Algorithm 1), an algorithm that finds keys for each class of a dataset. The instances of a given class are represented in a prefix tree (see Figure 3.2(a)). This structure is used to discover the sets of maximal undetermined keys and maximal non keys. Once all the undetermined keys and non keys are found, they are used to derive the set of minimal keys. KeyFinder repeats this process for every class of the given ontology. To compute keys that are valid for the classes of two ontologies, KeyFinder is applied in each dataset independently and once all the keys for every class are found, the obtained keys are then merged in order to compute the set of keys

that are valid for both datasets (see Figure 3.2(b)).

3.3 KD2R general algorithm

KeyFinder (see Algorithm 1) is the main algorithm of the KD2R which retrieves for each class of an RDF dataset that conforms to an OWL ontology, the set of minimal keys. KeyFinder sorts the classes by computing their topological order (see Line 1).

For each class, KeyFinder builds an *intermediate prefix tree* (see Line 5) which is a compact representation of the descriptions of instances of this class. Then, the *final prefix tree* (see Line 6) is generated in order to take into account the possible unknown property values. Using the final prefix tree and the set of inherited keys, if there exist any (see Line 11), UNKFinder (see Line 12) retrieves the maximal non keys and the maximal undetermined keys. Finally, to compute the complete set of minimal keys of a class, KeyFinder calls the KeyDerivation algorithm (see Line 13). In this section, KeyFinder exploits to the pessimistic heuristic. Section 3.4 provides information about the discovery of keys using the optimistic heuristic.

Algorithm 1: KeyFinder

Input : (in) RDF dataset D, Ontology O
Output: *Keys*: the set of minimal keys for each class c of O

1 $classList \leftarrow topologicalSort(O)$
2 **while** $classList \neq \emptyset$ **do**
3 c ← getFirst(classList) //*get and delete the first element*
$tripleList \leftarrow instanceDescriptions(c)$
4 **if** $tripleList \neq \emptyset$ **then**
5 $IPT \leftarrow$ createIntermediatePrefixTree($tripleList$)
6 $FPT \leftarrow$ createFinalPrefixTree(IPT)
7 $level \leftarrow 0$
8 $UK_{D.c} \leftarrow \emptyset$
9 $NK_{D.c} \leftarrow \emptyset$
10 $curUNKey \leftarrow \emptyset$
11 $inheritedKeys \leftarrow getMinimalKeys(Keys, c.superClasses)$
12 UNKFinder($FPT.root$, $level$, $inheritedKeys$, $UK_{D.c}$, $NK_{D.c}$, $curUNKey$)
13 $K_{D.c} \leftarrow$ KeyDerivation($UK_{D.c}$, $NK_{D.c}$)
$K_{D.c} \leftarrow getMinimalKeys(inheritedKeys.add(K_{D.c}))$
14 $Keys.c \leftarrow K_{D.c}$
15 **return** *Keys*

3.3.1 Illustrating example

We introduce an example that is going to be used throughout this chapter. In Figure 3.3, we present a part of DBpedia ontology concerning restaurants (name space db[1]). The class db:*Restaurant* is described by its name, its telephone number, its address and finally the city and the country where it is located. The class db:*Restaurant* is a subclass of the class db:*Building*. The Figure 3.4 contains RDF descriptions of four db:*Restaurant* instances.

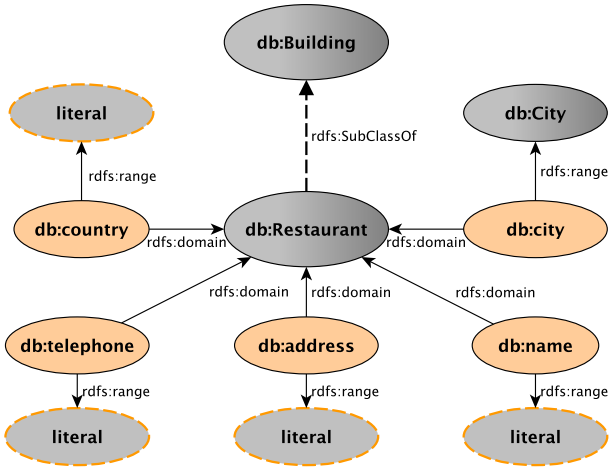


Fig. 3.3 A small part of DBpedia ontology for the class db:*Restaurant*

Dataset $D2$**:**
db:Restaurant(r1), $db{:}name(r1, "Arzak")$, $db{:}city(r1, c1)$, $db{:}country(r1, "Spain")$, $db{:}address(r1, "800\ Decatur\ Street")$,

db:Restaurant(r2), $db{:}name(r2, "Park\ \ Grill")$, $db{:}city(r2, c2)$, $db{:}country(r2, "USA")$, $db{:}address(r2, "11\ North\ Michigan\ Avenue")$,

db:Restaurant(r3), $db{:}name(r3, "Geno's\ Steaks")$, $db{:}country(r3, "USA")$, $db{:}telephone(r3, "884-4083")$, $db{:}telephone(r3, "884-4084")$, $db{:}address(r3, "35\ cedar\ Avenue")$,

db:Restaurant(r4), $db{:}name(r4, "joy\ Hing")$, $db{:}city(r4, c4)$, $db{:}country(r4, "China")$, $db{:}address(r4, "265\ Hennessy\ Road")$

Fig. 3.4 RDF dataset $D2$

[1] `http://dbpedia.org/ontology/`

3.3.2 Prefix Tree creation

We now describe the creation of the prefix tree which represents the descriptions of the instances of a given class in one dataset. Each level of the prefix tree corresponds to a property p and contains a set of *nodes*. Each node contains a set of *cells*. Each cell contains:

1. a cell value: (i) when p is a property, the cell value is one literal value, one URI instantiating its range or a null value and (ii) when p is an inverse property, the cell value is one URI instantiating its domain or an artificial null value.

2. *IL*: (i) when p is a property, the Instance List, called *IL*, is the set of URIs instantiating its domain and having as range the cell value, and (ii) when p is an inverse property, the Instance List is the set of URIs instantiating its range and having as domain the cell value.

3. *NIL*: the Null Instance List, called *NIL*, is the list of URIs for which the property value is unknown and for which we have assigned the cell value (null or not).

4. a pointer to a single child node.

Each prefix path corresponds to the set of instances that share cell values for all the properties involved in the path.

In order to consider the cases where property values are not given in the dataset, we create first an intermediate prefix tree, called *IP-Tree*. In *IP-Tree*, the absence of a value for a given property is represented by an artificial null value. The final prefix tree, called *FP-Tree*, is generated by assigning all the existing cell values of one node to the cell that contains the artificial null value.

3.3.2.1 *IP-Tree* creation

In order to create the *IP-Tree*, we use all the properties that appear at least in one description of an instance of the considered class. For each value of a property, if there does not exist already a cell value with the same value, a new cell is created and the Instance List *IL* is initialized with this instance. When a property does not appear in the description of an instance, we create or update, in the same way, a cell with an artificial null value. The creation of the *IP-Tree* is achieved by scanning the data only once.

Example 3.3.1. *Example of CreateIntermediatePrefixTree algorithm.*
Figure 3.5 shows the *IP-Tree* for the descriptions of instances of the class *db*:*Restaurant* in the RDF dataset *D*2 presented in Figure 3.4. The creation of the *IP-Tree* starts with the first

Algorithm 2: CreateIntermediatePrefixTree

Input : (in) RDF DataSet s , Class c
Output: *root* of the *IP-Tree*

```
1  root ← newNode()
2  P ← getProperties(c,s)
3  foreach c(i) ∈ s do
4      node ← root
5      foreach p_k ∈ P do
6          p_k(i) ← getValue(i)
7          if p_k(i) == ∅ then
8              if ∃ cell_1 ∈ node with null value then
9                  node.cell_1.IL.add(i)
10             else
11                 cell_1 ← newCell()
12                 node.cell_1.value ←null
13                 node.cell_1.IL.add(i)
14         else
15             foreach value v ∈ p_k(i) do
16                 if ∃ cell_1 ∈ node with value v then
17                     node.cell_1.IL.add(i)
18                 else
19                     cell_1 ← newCell()
20                     node.cell.value ← v
21                     node.cell.IL.add(i)
22         if p_k is not the last property then
23             if hasChild(cell_1) then
24                 node ← cell.child.node()
25             else
26                 node ← cell.child.newNode()
27 return root
```

instance which is the restaurant $r1$. A new cell is created in the root node containing the name of the country in which the restaurant is located. The next information concerning this restaurant is the city where it is located. To store this information a new node will be created as a child node of the cell *"Spain"*. In this new node, a new cell is created to store the value $c1$. The process continues until all the information about an instance are represented in the tree. For each new instance, the insertion begins again from the root.

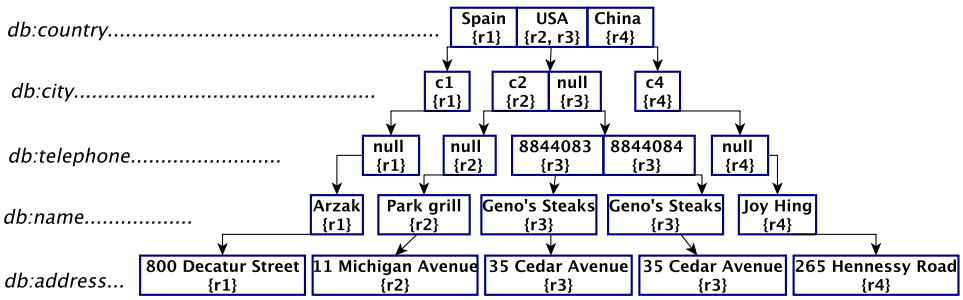


Fig. 3.5 *IP-Tree* for the instances of the class *db*:*Restaurant*

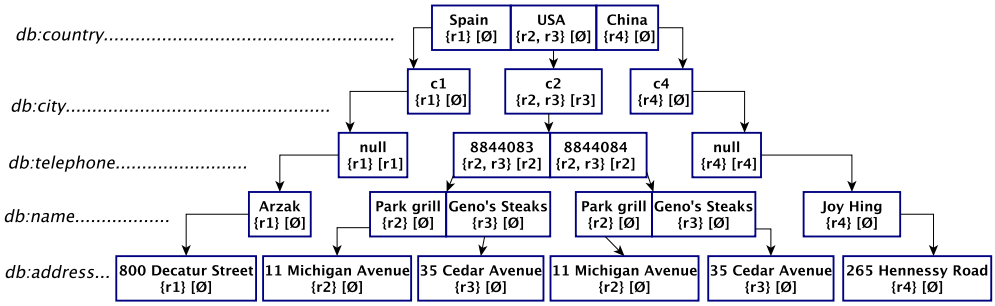


Fig. 3.6 *FP-Tree* for the instances of the class *db*:*Restaurant*

3.3.2.2 Final Prefix Tree creation

Using the *IP-Tree*, we generate a *FP-Tree* (see Algorithm 3). This is done by assigning, for each node, the set of possible values contained in its cells, to the artificial null value of this node. If no null values exist in an *IP-Tree*, this tree is also the *FP-Tree*. We use the Null Instance List *NIL* to store the instances for which the property value is unknown. This information will be used by UNKFinder (Algorithm 5) to distinguish non keys from undetermined keys.

Example 3.3.2. *Example of CreateFinalPrefixTree algorithm.*
In Figure 3.6, we give the *FP-Tree* of the RDF dataset $D2$. As we notice in Figure 3.5, the restaurants $r2$ and $r3$ are both located in *"USA"*. The restaurant $r2$ is located in the city $c2$ while there is no information about the location of the restaurant $r3$. This absence is represented by a null cell in the *IP-Tree*. Therefore, to build the *FP-Tree*, we assign the value $c2$ to null value of $r3$ for the property *db*:*city*. The *NIL* is now $\{r2, r3\}$ and $r3$ is stored in *NIL* (see Figure 3.7(b)). This assignment is performed using the *mergeCells*

operation. This process is applied recursively to the children of this node (see Figure 3.7(c)) in order to: (i) merge the cells of the child nodes that contain the same value and (ii) to replace the null values by the remaining possible ones.

Algorithm 3: CreateFinalPrefixTree

Input : IPT: *IP-Tree*
Output: FPT: *FP-Tree*

```
1 FPT.root ← mergeCells(getCells(IPT.root))
2 foreach cell c ∈ FPT.root do
3     nodeList ← getSelectedChildren(IPT.root, c.value)
4     nodeList.add(getSelectedChildren(IPT.root, null))
5     c.child ← MergeNodeOperation(nodeList)
6 return FPT
```

Algorithm 4: MergeNodeOperation

Input : (in) *nodeList*, a list of *nodes* to be merged
Output: *mergedNode*, the merged node and its descendants

```
1 cellList ← getCells(nodeList)
2 mergedNode ← mergeCells(cellList)
3 if nodeList contains non leaf nodes then
4     foreach cell c ∈ mergedNode do
5         childrenNodeList.add(getSelectedChildren(nodeList, null))
6         childrenNodeList.add(getSelectedChildren(nodeList, c.value))
7         c.child ← MergeNodeOperation(childrenNodeList)
8 return mergedNode
```

3.3.3 Undetermined and non key discovery

The UNKFinder algorithm is used to discover the sets of undetermined keys and non keys. UNKFinder aims at retrieving the set of maximal undetermined keys $UK_{D.c}$ and the set of maximal non keys $NK_{D.c}$ from the *FP-Tree*. For this purpose, this algorithm searches the biggest set of properties for which there exist at least two instances sharing values. To do so, the *FP-Tree* is traversed in a depth-first way. When a set of properties is found, it represents either a non key or an undetermined key.

A set of properties is added to $NK_{D.c}$ or to the $UK_{D.c}$ only when a leaf node (i.e., a node found in the last level of a tree) is reached. More precisely, when a leaf node is reached, if

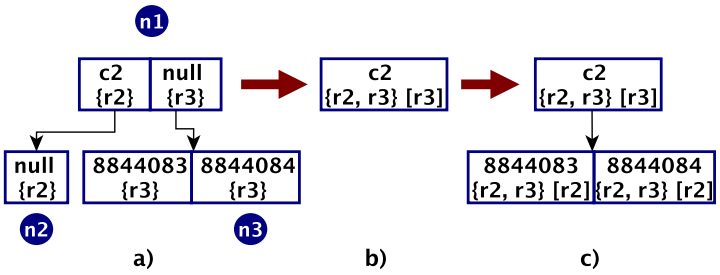


Fig. 3.7 Example of MergeNodeOperation

one of its cells contains an *IL* with size bigger than 1, we are sure that the constructed set of properties (*curUNKey*) is either a non key or an undetermined key. If one of the instances found in the *IL* is obtained by a merge with a null value, then *curUNKey* is an undetermined key, otherwise it is a non key.

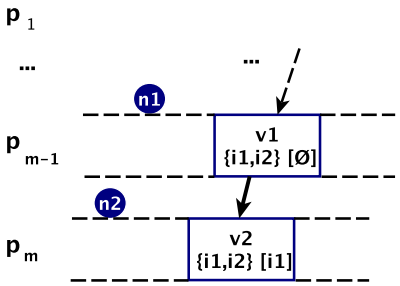


Fig. 3.8 Set of properties $\{p_1 \dots p_m\}$ representing an undetermined key

The set of properties $\{p_1, \dots, p_m\}$ of the Figure 3.8 represents an undetermined key, since the instances i1 and i2 share a value for each property in $\{p_1, \dots, p_m\}$ and the cell of p_m comes from a merge with a null cell.

Additionally, when the size of the union of all the *IL* of the leaf node is greater than 1, we know that the *curUNKey*, constructed before adding the leaf level, is a non key or an undetermined key (same criteria as presented above to distinguish them).
For example, in Figure 3.9, we notice that the set of properties $\{p_1, \dots, p_{m-1}\}$ is either a

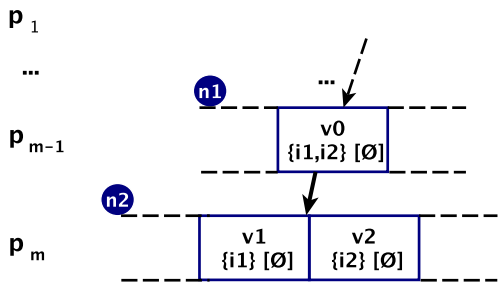


Fig. 3.9 Set of properties $\{p_1 \dots p_{m-1}\}$ representing a non key or an undetermined key

non key or an undetermined key since there exists more than one cell in the leaf node $n2$, i.e., $|\{i_1\} \cup \{i_2\}| > 1$.

In order to discover all the non keys and undetermined keys in the data, we should generate all the possible combinations of properties. To do so, we need to ignore some level(s) in the prefix tree. Therefore, the child nodes of the ignored level(s) have to be merged using the MergeNodeOperation (see Algorithm 4).

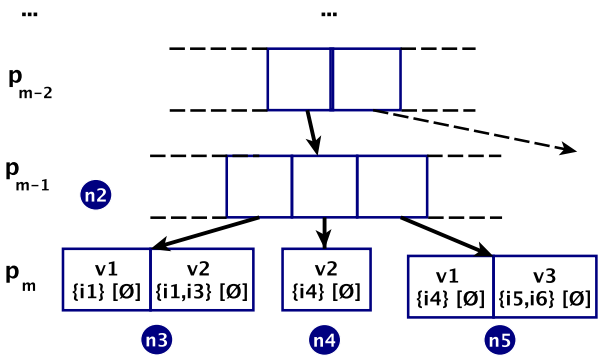


Fig. 3.10 Prefix tree before the suppression of p_{m-1}

Let us consider the prefix tree of the Figure 3.10. Once the set of properties $\{p_1, \dots, p_{m-1}, p_m\}$ is tested successively on the leaf nodes n_3, n_4 and n_5, the property p_{m-1} is suppressed. Using the MergeNodeOperation (see Algorithm 4), applied on the children of n_2, the new prefix tree shown in the Figure 3.11 is constructed. In this tree, the new node n_6 represents the result of the MergeNodeOperation on n_3, n_4 and n_5. When needed, this operation is reapplied recursively on the new prefix trees obtained from the merge.

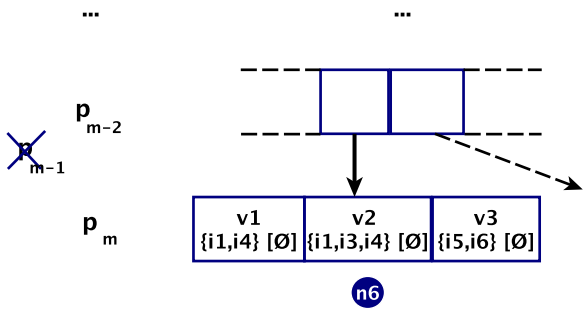


Fig. 3.11 Result of the MergeNodeOperation applied on *FP-Tree* of the Figure 3.10

3.3.3.1 Pruning strategies

To ensure the scalability of the undetermined and non key discovery, UNKFinder performs three kinds of pruning strategies.

Key pruning. The subsumption relation between classes is exploited to prune the prefix tree traversal. Indeed, when a key is already discovered for a class using one dataset, then this key is also valid for all the subclasses in this dataset. Thus, parts of the prefix tree are not explored.

For example, we consider $K_{D.c1} = \{\{p_1,\ p_3\},\ \{p_2,\ p_4\}\}$ the set of keys of the class c_1. Let c_2 be a subclass of c_1 in the ontology. This means that all the keys of the class c_1 will be also valid keys for the class c_2. Indeed, if a key is valid in a dataset (i.e., descriptions of a superclass), it will be also valid in every subset of this dataset (i.e., descriptions of a subclass). Let us now consider the prefix tree of the class c_2 shown in Figure 3.12. When the $curUNKey = \{p_1,\ p_2,\ p_3\}$, the pruning is applied since the *curUNKey* includes one of the keys of c_1 (i.e., $\{p_1,\ p_3\}$). In this case, $\{p_1,\ p_2,\ p_3\}$ is a key as well. Therefore, the subtree rooted at n_3 will not explored.

Antimonotonic pruning. This strategy exploits the anti-monotonic characteristic of a non key, i.e., if a set of properties is a non key, all its subsets are, by definition, non keys. Thus, no subset of an already discovered non key will be explored. In other words, when all the new combinations of properties in a given path cannot lead to new maximal non keys, then the exploration of this path stops.

For example, let $NK_{D.c} = \{\{p_1,\ p_2,\ p_3\}\}$ be the set of already discovered non keys. Suppose that $curNKey = \{p_1\}$. If the remaining levels of the prefix tree correspond only to

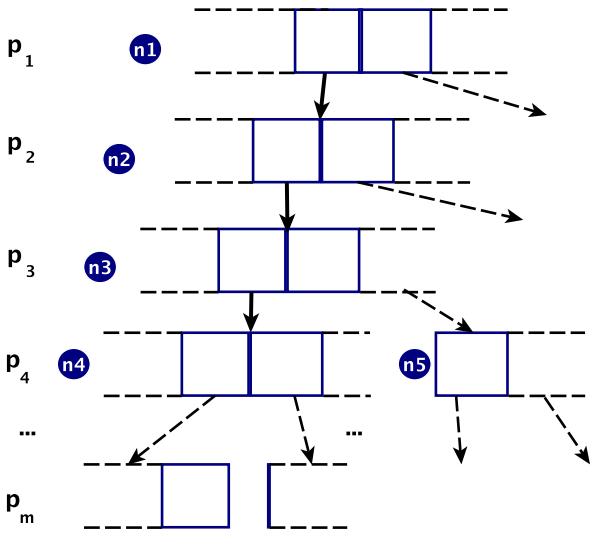


Fig. 3.12 Pruning paths of a prefix tree when a key is known

the properties p_2 and/or p_3, then the children of the current node are not explored.

Monotonic pruning. This strategy exploits the monotonic characteristic of keys, i.e., if a set of properties is a key then all the supersets of this key are also keys. Therefore, when a node describes only one instance, i.e., it contains only one cell, we are sure that no non key can be found using this node. Thus, if while traversing the prefix tree a node is only composed of one cell, the current path will not be explored since it cannot lead to a non key.

3.3.3.2 UNKFinder algorithm

In order to discover all the maximal non keys, UNKFinder (see Algorithm 5) traverses the prefix tree.

The algorithm takes the following inputs: (i) *root* a node of the prefix tree, (ii) *level* a number assigned to the current property, (iii) *inheritedKeys* the set of keys inherited from all the super-classes of the current class, (iv) $UK_{D.c}$ the set of undetermined keys, (v) $NK_{D.c}$ the set of non keys and finally (vi) *curUNKey* the candidate non key or undetermined key.

UNKFinder is called for the first time with the following parameters: UNKFinder(*root*, 1, *inheritedKeys*, ∅, ∅, ∅) where *root* is the node that corresponds to the root of the prefix tree and *inheritedKeys* the set of already known keys, if any.

The algorithm begins from the root of the prefix tree and makes a depth-first traversal of it in order to discover the maximal undetermined keys and non keys.

In each iteration of the algorithm, the variable *curUNKey* represents the current candidate undetermined key or non key. The *curUNKey* contains the identifiers of the properties that are currently explored. Let us consider the case where the *root* node does not correspond to a leaf (see Line 18). The size of the *IL* of each cell is checked only if the following conditions are satisfied: (i) the cells of the *root* do not represent values of only one instance (see Line 20), (ii) the *curUNKey* does not belong to an already known key (see Line 12) and (iii) there exist new candidate maximal non keys. If the size of the *IL* is bigger than one, the UNKFinder is called for the next level. When all the cells of the *root* have been explored, the *level* of the *root* is removed from the *curUNKey*, the children of the node are merged and the UNKFinder is executed on the new merged tree *mergedTree*.

When UNKFinder proceeds to a leaf, if the *IL* size of one cell is bigger than 1, this means that there are more than one instances with the same cell values in the prefix path and the *curUNKey* will be added to either the $NK_{D.c}$ or to the $UK_{D.c}$. In order to be able to separate the non keys from the undetermined keys, we have to check if one of the cells that participates in the *curUNKey* has been obtained by a merge with a null value. If so, the *curUNKey* will be added in the set $UK_{D.c}$. The algorithm continues by removing the current *level* from the *curUNKey*. If the current root has more than one cell and at least one of these cells has *IL* bigger than 1, the *curUNKey* will be added to either $NK_{D.c}$ or $UK_{D.c}$.

Example 3.3.3. *Example of UNKFinder algorithm.*
We illustrate UNKFinder algorithm on the *FP-Tree* shown in Figure 3.6. The algorithm is called for the first time with the root node of the prefix tree. In this case, the *level* 1 is added to the *curUNKey*. The first cell to be explored contains the value *"Spain"*. Since the *IL* of this cell has size one (r1), the algorithm will not examine the children of this cell (see Line 26). Indeed, when a cell has *IL* of size 1 this means that it describes only one instance. Therefore, no undetermined key or non key can be found.

The algorithm moves to the next cell of the root node, containing the value *"USA"*. Since the list *IL* has size 2, UNKFinder is called for the child node of this cell. Now the *curUNKey* is $\{1,2\}$ and the current cell is‘*"c2"*. The list *IL* of this cell has size bigger than 1 thus, UNKFinder is now called for the child node of the *"c2"*. The *level* of the property *db:telephone* is added to the *curUNKey*, i.e., *curUNKey* = $\{1, 2, 3\}$. Starting from the

Algorithm 5: UNKFinder

```
Input : (in) root: node of the prefix tree
        (in) level: property number
        (in) inheritedKeys: keys inherited from super-classes
        (in/out) UK_{D.c}: set of undetermined keys
        (in/out) NK_{D.c}: set of non keys
        (in/out) curUNKey_{D.c}: candidate undetermined or non key
1  curUNKey.add(level)
2  if root is a leaf then
3      foreach cell c ∈ root do
4          if c.IL.size() > 1 then
5              if one of the cells of the prefix path comes from a merge with null value (NIL.size()>1)
               then UK_{D.c}.add(curUNKey)
6
7              else
8                  NK_{D.c}.add(curUNKey)
9                  UK_{D.c}.delete(curUNKey)
10                 break
11     curUNKey.remove(level)
12     if root has more than one cell and union(getIL(root.cells))).size() > 1 then
13         if one of the cells of the prefix path comes from a merge with null value (NIL.size()>1) then
14             UK_{D.c}.add(curUNKey)
15         else
16             NK_{D.c}.add(curUNKey)
17             z
18 else
19     //pruning: monotonic characteristic of keys (curUNKey is a key for the current path)
20     if IL of each cell of root contains the same instances then
21         return
22     //pruning: monotonic characteristic of inherited keys and anti-monotonic characteristic of non
       keys
23     if ∄ k ⊆ curUNKey s.t. k ∈ inheritedKeys and new maximal non keys are achievable through the
       current path then
24         foreach cell c ∈ root do
25             //pruning: monotonic characteristic of keys
26             if c.IL.size() > 1 then
27                 UNKFinder(c.getChild,level+1,inheritedKeys, UK_{D.c}, NK_{D.c} )
28     curUNKey.remove(level)
29     //pruning: anti-monotonic characteristic of non keys
30     if new maximal non keys are not achievable through the current path then
31         return
32     childNodeList ← getChildren(root)
33     mergedTree ← MergeNodeOperation(childNodeList)
34     UNKFinder(mergedTree,level+1, inheritedKeys, UK_{D.c}, NK_{D.c})
```

cell *"88844083"* we observe that the list *IL* has size bigger than 1. Thus, UNKFinder is called for the child node of this cell. The *curUNKey* is now $\{1, 2, 3, 4\}$. We observe that both cells *"Park Grill"* and *"Geno's Steaks"* have *IL* of size 1. This means that the UNKFinder will not be called with any of these cells at this step. Proceeding to the Line 11 of the algorithm, the *curUNKey* is now $\{1, 2, 3\}$. The children of this node are merged and UNKFinder is called with the new merged node called *mergedTree*. The *curUNKey* is equal to $\{1, 2, 3, 5\}$. As we can see, the node *mergedTree* corresponds to a leaf node. Since none of the cells of this node have *IL* with size bigger than 1, the *level* 5 is removed from the *curUNKey*. Now the *curUNKey* contains 1, 2 and 3 (see Line 1). Since (i) the root node contains more than one cell, (ii) the cells refer to more than one instances (see Line 12) and (iii) there exists at least one cell that comes from the merge with a null value (see Line 13), the set $\{1, 2, 3\}$ is added to the $UK_{D.c}$.

We now proceed to the cell *"88844084"*. Since the list *IL* has size bigger than 1, the UNKFinder is called for the child node of this cell. The *curUNKey* is again $\{1, 2, 3, 4\}$. We observe that both cells have a *IL* of size 1, thus UNKFinder will not be called. As before, the children of this node are merged and the UNKFinder is called for the new merged node. The *curUNKey* is $\{1, 2, 3, 5\}$. Since all the cells have *IL*s of size 1, the *curUNKey* is now $\{1, 2, 3\}$. We notice that the *uk* $\{1, 2, 3\}$ is already contained in the $UK_{D.c}$. Now the *level* 3 is removed from the *curUNKey* and the children of the two cells *"88844083"* and *"88844084"* are merged. Calling UNKFinder with the new merged node, the *curUNKey* is now $\{1, 2, 4\}$. Since both cells have *IL*s of size 1, the *level* 4 is removed from the *curUNKey*. The children of this node are now merged and *curUNKey* is $\{1, 2, 5\}$. The current node corresponds to a leaf node. Both cells *"11 Michigan Avenue"* and *"35 Cedar Avenue"* have *IL*s of size 1. Thus, the *level* 5 is removed from the *curUNKey*.

Since this node is composed of more than one cells and there exists at least one cell that comes from the merge with a null value, the set $\{1, 2\}$ is added to the $UK_{D.c}$.

Continuing these steps, we obtain the following sets of maximal undetermined keys and maximal non keys for the class *db*:*Restaurant*:

$UK_{D2.db:Restaurant} = \{\{db{:}telephone,\ db{:}city,\ db{:}country\}\}$

$NK_{D2.db:Restaurant} = \{\{db{:}country\}\}$.

3.3.3.3 Key derivation

Once the sets of maximal undetermined keys and maximal non keys of one class are discovered for a given dataset, we are able to derive the set of minimal keys. The main idea

is that a key is a set of properties that is not included or is not equal to any maximal non key or undetermined key. To build all these sets of properties, for each maximal non key and undetermined key, we retain the properties that do not belong to this non key or undetermined key. Then, the Cartesian product of the obtained properties is computed and only the minimal sets are kept.

More precisely, as shown in the Algorithm 6 inspired from [SBHR06], to derive the minimal keys $K_{D.c}$, we first compute the union of $NK_{D.c}$ and $UK_{D.c}$ and select only the maximal sets of properties (see Line 2). For each selected set of properties, we compute the complement set with respect to the whole set of instantiated properties. The Cartesian product of all the complement sets gives all the combinations of properties that are not non keys, thus that are keys. Once all the keys are found, the function *getMinimalKeys* is applied to remove all non-minimal keys from the obtained set $K_{D.c}$.

Algorithm 6: KeyDerivation

```
   Input  : UK_{D.c}: set of maximal undetermined keys
            NK_{D.c}: set of maximal non keys
   Output: K_{D.c}: set of minimal keys
1  K_{D.c} ← ∅
2  UNK_{D.c} ← getMaximalUNKeys(UK_{D.c} ∪ NK_{D.c})
3  foreach unk ∈ UNK_{D.c} do
4      complementSet ← complement(unk)
5      if K_{D.c} == ∅ then
6          K_{D.c} ← complementSet
7      else
8          newSet ← ∅
9          foreach p_k ∈ complementSet do
10             foreach k ∈ K_{D.c} do
11                 newSet.insert(k.add(p_k))
12         newSet ← getMinimalKeys(newSet)
13         K_{D.c} ← newSet
14 return K_{D.c}
```

For example, in the class *db*:*Restaurant* we have discovered the set of undetermined keys $UK_{D2.db:Restaurant} = \{\{db:telephone,\ db:city,\ db:country\}\}$ and the set of non keys $NK_{D2.db:Restaurant} = \{\{db:country\}\}$.
In this case, there exists only one maximal set of properties:
$\{\{db:telephone, db:city, db:country\}\}$.
The complement set of $\{db:telephone, db:city, db:country\}$ is:

$\{db{:}address,\ db{:}name\}$.
Since there is only one set of properties, we obtain the following minimal keys:
$K_{D2.db:Restaurant} = \{\{db{:}address\}, \{db{:}name\}\}$.

3.3.3.4 Valid keys in different datasets

Given two datasets that conform to different ontologies, and the sets of discovered keys in each one of them, we propose a strategy that computes keys that are valid in both datasets. We consider that all the mappings between the two ontologies are available. The keys are expressed using common vocabulary. The process starts by deleting from $K_{D.c}$ all the keys that contain properties that do not belong to the set of mapped properties called $\mathscr{P}e_{ic}$. Indeed, if a key contains properties that are not used by the other ontology, this key can never be valid for both ontologies. Considering two equivalent classes and their sets of keys, the objective is to find the smallest sets of properties that are valid keys in both datasets. For this purpose, we compute the Cartesian product of their minimal keys. As a final step, we select only the minimal ones. Following these steps, we guarantee that the obtained keys are valid in both datasets.

For example, consider two datasets $D = \{D2, D3\}$. If the sets of minimal keys in the dataset $D2$ and in the dataset $D3$ are:
$K_{D2.db:Restaurant} = \{\{db{:}address\}, \{db{:}name\}\}$ and
$K_{D3.db:Restaurant} = \{\{db{:}telephone,\ db{:}city\}, \{db{:}name\}\}$
then the valid keys will be:
$K_{D:Restaurant} = \{\{db{:}telephone,\ db{:}address,\ db{:}city\}, \{db{:}name\}\}$.

3.4 Key Discovery applying the Optimistic heuristic

To consider the optimistic heuristic, it suffices to call the KeyDerivation (see Algorithm 6) only with the set of non keys $NK_{D.c}$ instead of calling it with the union of the sets of non keys and undetermined keys ($NK_{D.c} \cup UK_{D.c}$).

In the case where the undetermined keys are not necessary and their computation can be avoided, a more efficient key discovery method can be performed. Indeed, considering that each null value can be one of the already existing ones, means that we have to assign all the values to each not given one. This makes the approach based on the pessimistic heuristic not scalable when the data are incomplete. As we have already mentioned, the pessimistic heuristic considers that missing values can be any of the already existing values appearing

in the data. Using this approach many keys can be lost due to data incompleteness. For example, let us consider a dataset describing 1000 people and one of its properties, the property *mobilePhone*, which is given for 999 people in this dataset. Even when all the 999 values of the property *mobilePhone* are pairwise distinct for each person, *mobilePhone* will not be discovered when the pessimistic heuristic is applied. on the contrary, applying the optimistic heuristic, the property *mobilePhone* will be discovered as key since the absence of the values is not taken into account.

The keys discovered applying the optimistic heuristic, i.e., *optimistic keys*, correspond to the union of the keys (see Definition 7) and the undetermined keys (see Definition 11).

Definition 12. (Optimistic Key). *A set of properties $P = \{p_1, \ldots, p_n\}$ ($P \subseteq \mathscr{P}$) is an optimistic key for the class c ($c \in \mathscr{C}$) in D if:*

$$\forall X \; \forall Y \; ((X \neq Y) \wedge c(X) \wedge c(Y)) \Rightarrow$$

$$(\forall Z \; \neg(p_j(X,Z) \wedge p_j(Y,Z)))$$

For example, the property *db*:*address* is an optimistic key for the class *db*:*Restaurant* since the addresses of all the restaurants appearing in the dataset $D2$ are distinct.

We denote $K_{D.c}$ the set of optimistic keys of the class c w.r.t the dataset D.

From now on, we will refer to what has been presented in the Section 3.3 as KD2R pessimistic and what we presented in this section as KD2R optimistic. Using the optimistic heuristic, there is no more need for a *IP-Tree* (see Algorithm 2). Using Algorithm 7, we can build directly the *FP-Tree*, since no null values have to be merged. To discover only the optimistic keys using the general Algorithm 1, we just replace the steps of creation of the *IP-Tree* and the creation of the *FP-Tree* by the creation of the optimistic prefix tree.

3.5 Experiments

In this section, we present the results of the experiments obtained applying KD2R in different datasets. The experiments are grouped in three categories. In Section 3.5.1, we provide the obtained keys for each dataset. Section 3.5.2 shows the scalability of KD2R when the optimistic heuristic is applied. Finally, the evaluation of the quality of the discovered keys is performed in Section 3.5.3, using the data linking tool N2R. The quality of the data linking is measured using the recall, the precision and the F-measure.

Algorithm 7: createOptimisticPrefixTree

```
Input  : RDF DataSet s ,
         Class c
Output: root of the optimistic prefix tree
1  root ← newNode()
2  P ← getProperties(c, s)
3  foreach c(i) ∈ s do
4      node ← root
5      for each p_k ∈ P do
6          p_k(i) ← getValue(i)
7          foreach value v ∈ p_k(i) do
8              if there exists a cell cell_1 with value v then
9                  node.cell_1.IL.add(i)
10             else
11                 cell_1 ← newCell()
12                 node.cell.value ← v
13                 node.cell.IL.add(i)
14         if p_k is not the last property then
15             if hasChild(cell_1) then
16                 node ← cell.child.node()
17             else
18                 node ← cell.child.newNode()
19 return root
```

The experiments have been executed on a single machine with 4GB of RAM, processor Intel(R) Core(TM) i5 CPU 650@3.20GHz running Windows 7 (64-bit).

3.5.1 Discovered keys in different datasets

We have evaluated KD2R on thirteen RDF datasets[2]. Four datasets have been used in OAEI 2010 (Ontology Alignment Evaluation Initiative)[3], in the Instance Matching track. Two more datasets have been taken from the OAEI 2011[4]. Four datasets have been collected from the Web of data. In the context of the Qualinca project [5], three additional datasets have been used for the experimental evaluation. Each dataset conforms to an OWL ontology.

[2] http://www.lri.fr/~sais/KD2R-DataSets
[3] http://oaei.ontologymatching.org/2010/
[4] http://oaei.ontologymatching.org/2011/
[5] http://www.lirmm.fr/qualinca/?q=en/en/home

UNA is declared for some of these datasets. For each dataset, we discover the keys applying both the optimistic and pessimistic heuristics.

Table 3.1 displays some statistics of the used datasets. The table contains the number of triples, the number of instances per class and the number of properties per class. In the following, we describe each dataset and present the set of keys that are discovered by KD2R.

Dataset	#triples	#instances (per class)	#properties (per class)
OAEI:Person1 - D1	5801	*Person*: 500	*Person*: 7
		Address: 500	*Address*: 6
OAEI:Person2 - D2	6230	*Person*: 500	*Person*: 7
		Address: 500	*Address*: 6
OAEI:Restaurant1 - D3	891	*Restaurant*: 113	*Restaurant*: 4
		Address: 113	*Address*: 3
OAEI:Restaurant2 - D4	3347	*Restaurant*: 752	*Restaurant*: 4
		Address: 752	*Address*: 3
GFT - D5	4494	*Restaurant*: 1349	*Restaurant*: 4
ChefMoz - D6	153300	*Restaurant*: 32686	*Restaurant*: 4
OAEI:Film1 - D7	117076	*Film*: 1288	*Film*: 8
		Creature: 8401	*Creature*: 9
		Location: 2471	*Location*: 8
		Language: 67	*Language*: 4
		Budget: 101	*Budget*: 2
OAEI:Film2 - D8	129429	*Film*: 1288	*Film*: 8
		Creature: 8401	*Creature*: 9
		Location: 2471	*Location*: 8
		Language: 67	*Language*: 4
		Budget: 101	*Budget*: 2
DB:Person - D9 (T=20%)	2952706	740689	7
DB:Person - D9 (T=10%)	3332207	742233	10
DB:NaturalPlace - D10 (T=20%)	836960	49887	11
INA - D11 (T=30%)	596415	*Contenu*: 44779	*Contenu*: 82
		Personne: 7444	*Personne*: 44
ABES1 - D12	59839	*Person*: 5671	*Person*: 9
ABES2 - D13	8738	*Person*: 573	*Person*: 13

Table 3.1 Statistics on the used datasets

3.5.1.1 KD2R results on OAEI 2010 datasets

Datasets *D1*, *D2*. Both datasets $D1$ and $D2$, provided by the Instance Matching track of OAEI 2010, contain 1000 instances of the classes *Person* and *Address* (see Table 3.1). In their common ontology:

- a *Person* instance is described by the datatype properties *givenName*, *state*, *surname*, *dateOfBirth*, *socSecurityId*, *phoneNumber*, *age* and the object property *hasAddress*.
- an *Address* instance is described by the datatype properties *street*, *houseNumber*, *postCode*, *isInSuburb*[6] and the object property *hasAddress*.

Both RDF datasets $D1$ and $D2$ contain each 500 instances of the class *Person* and 500 instances of the class *Address*.

KD2R has discovered the following four keys that are valid in both $D1$ and $D2$ for the classes *Person* and *Address*, using the pessimistic heuristic:
$K_{D1D2.Person} = \{\{socSecurityId\}, \{hasAddress\}\}$
$K_{D1D2.Address} = \{\{isInSuburb, postcode, houseNumber\}, \{inv\text{-}hasAddress\}\}$.

Applying the optimistic heuristic, KD2R has discovered the following thirteen keys that are valid in $D1$ and $D2$, for the classes *Person* and *Address*:
$K_{D1D2.Person} = \{\{socSecurityId\}, \{hasPhone\}, \{hasAddress\}, \{dateOfBirth, givenName\}, \{dateOfBirth, age\}, \{surname, dateOfBirth\}, \{surname, givenName\}\}$
$K_{D1D2.Address} = \{\{street, houseNumber\}, \{street, isInSuburb\}, \{houseNumber, isInSuburb\}, \{postCode, isInSuburb\}, \{street, postCode\}, \{inv\text{-}hasAddress\}\}$.

Using the optimistic heuristic, all the undetermined keys are considered as keys. Both datasets $D1$ and $D2$ contain a lot of not instantiated properties for the class *Person*. Thus, we have obtained a significant number of undetermined keys. This has led to a set of keys that is much bigger than the one obtained using the pessimistic heuristic.

Datasets *D3*, *D4*. The datasets $D3$ and $D4$ contain descriptions of the classes *Restaurant* and *Address* (see Table 3.1). Both datasets correspond to the first version of the OAEI 2010 restaurant dataset that contains bugs. In the provided ontology:

[6] in the ontology of the second dataset *isInSuburb* is declared as an object property. Since it was the unique difference between the two ontologies, we have chosen to rewrite the second dataset using the first ontology.

- a *Restaurant* instance is described using the datatype properties properties *name*, *phoneNumber*, *hasCategory* and the object property *hasAddress*.

- an *Address* instance is described using the datatype properties *street*, *city* and the object property *hasAddress*.

The dataset $D3$ contains 113 *Address* instances and 113 *Restaurant* instances while the dataset $D4$ contains 752 *Restaurant* instances and 752 *Address* instances.

The five valid keys in $D3$ and $D4$ obtained under the pessimistic heuristic, are as follows:
$K_{D3D4.Restaurant} = \{\{phoneNumber, name\}, \{phoneNumber, hasCategory\}, \{hasAddress\}, \{name, hasCategory\}\}$
$K_{D3D4.Address} = \{\{inv\text{-}hasAddress\}\}$.

Since $D3$ and $D4$ do not contain any undetermined key, the obtained results are the same for both optimistic and pessimistic heuristics.

3.5.1.2 KD2R results on GFT-ChefMoz datasets

Datasets *D5*, *D6*. The dataset $D5$ contains data extracted from the ChefMoz repository published on the LOD, while the dataset $D6$ contains data found in Google Fusion tables service [GHJ+10], by [QSSR12]. Each dataset conforms to a distinct OWL ontology.
The *GFT* dataset $D5$ contains 1349 instances of the class *Restaurant* (see Table 3.1). In the ontology, a restaurant is described by the datatype properties *title*, *address*, *cuisine* and *city*.
The *ChefMoz* dataset $D6$ contains 32686 instances of the class *Restaurant* (see Table 3.1). This dataset has been cleaned to remove duplicate instances. To do so, instances with similar names have been linked and manually checked in order to suppress duplicates. In this ontology, restaurants are described using more properties than in ontology of the dataset $D5$. Equivalence mappings have been set for the properties of *GFT* $D5$ and *ChefMoz* $D6$.
KD2R has discovered the following keys for the class *Restaurant* in the dataset $D5$, using the pessimistic heuristic:
$K_{D5.Restaurant} = \{\{address\}, \{city, title\}\}$.

The key that is obtained for *Restaurant* in the dataset $D6$ is the following composite key, using the pessimistic heuristic:

$K_{D6.Restaurant}$= $\{\{title, address\}\}$.

After the merge, the obtained key is:
$K_{D5D6.Restaurant} = \{\{title, address\}\}$.

Using the optimistic heuristic, the keys obtained on each dataset are different but the key obtained after their merge is the same as the one obtained using the pessimistic heuristic.

3.5.1.3 KD2R results on OAEI 2011 datasets

Datasets *D7*, *D8*. We now present the results of KD2R in IIMB (ISLab Instance Matching Benchmark) datasets $D7$ and $D8$. This benchmark is used in the instance matching track of the Ontology Alignment Evaluation Initiative (OAEI 2011 & 2012). An initial dataset $D7$ describing movies (films, actors, directors, etc.) is extracted from the Web (file 0). The classes of this dataset are $Film$, $Creature$, $Language$, $Budget$ and finally $Location$. In their common ontology:

1. A $Film$ instance is described by the datatype properties *name* and $estimatedBudgetUsed$ and the object properties $filmedIn$, $directedBy$, $starringIn$, $shotIn$, $article$ and $featuring$.

2. A $Creature$ is described by the datatype properties $bornIn$, *name*, $gender$, $article$, $dateOfBirth$ and $religion$ and the object properties $featuring$, $createdBy$ and $actedBy$.

3. A *Language* is described by the datatype property $iso639Code$ and the object properties $spokenIn$, $mainlySpokenIn$ and $dialect$.

4. A *Budget* is described by the datatype properties *currency* and *amount*.

5. A $Location$ is described by the datatype properties *name*, $formOfGovernment$, $callingCode$, $article$, *currency* and *size* and the object properties $hasCity$ and $hasCapital$.

Various kinds of transformations, including structural, logical and value transformations, were applied to this initial dataset to generate a set of 80 different test cases. For each test case, the complete set of *owl:sameAs* links between individuals of the generated test case and the ones of the initial dataset, is given. We evaluate the discovered keys using the first

test case *D*8 in which the modifications only concern datatype property values (typographical errors, lexical variations). Each of the two datasets contain 1228 descriptions of films, 8401 descriptions of creatures, 2471 descriptions of locations, 67 descriptions of languages and finally 101 different descriptions of budgets concerning the movies.

In this experiment we present only the results of the optimistic heuristic, since the pessimistic heuristic cannot scale. The keys found for each class are presented in Table 3.2. Note that no key has been found for the class *Budget*.

Creature	***Location***
{{*actedBy*, *religion*},	{{*currency*, *hasCity*, *callingCode*},
{*name*, *actedBy*, *gender*},	{*hasCity*, *formOfGovernment*, *callingCode*},
{*dateOfBirth*, *name*, *religion*},	{*currency*, *formOfGovernment*, *callingCode*},
{*name*, *featuring*, *gender*},	{*name*, *hasCapital*},
{*createdBy*, *featuring*},	{*currency*, *callingCode*, *hasCapital*},
{*name*, *createdBy*},	{*currency*, *name*},
{*article*},	{*hasCity*, *size*},
{*dateOfBirth*, *bornIn*, *religion*},	{*name*, *callingCode*},
{*dateOfBirth*, *gender*, *religion*},	{*callingCode*, *size*},
{*dateOfBirth*, *featuring*},	{*currency*, *size*},
{*featuring*, *bornIn*},	{*hasCapital*, *size*},
{*bornIn*, *actedBy*},	{*formOfGovernment*, *size*},
{*createdBy*, *actedBy*},	{*article*},
{*createdBy*, *religion*},	{*name*, *formOfGovernment*},
{*featuring*, *religion*},	{*name*, *hasCity*},
{*dateOfBirth*, *actedBy*},	{*formOfGovernment*, *callingCode*, *hasCapital*}}
{*createdBy*, *bornIn*},	
{*name*, *gender*, *religion*},	
{*dateOfBirth*, *createdBy*}}	
Film	***Language***
{{*article*},	{{*dialect*},
{*estimatedBudgetUsed*},	{*iso*639*Code*}}
{*filmedIn*, *directedBy*, *name*}}	

Table 3.2 Discovered keys for the datasets *D*7, *D*8

3.5.1.4 KD2R results on DBpedia datasets

In the case of the DBpedia dataset we provide only results found using the optimistic heuristic, since the pessimistic heuristic does not scale.

Dataset	*DB*:*Person* (*D*9) 20%	*DB*:*Person* (*D*9) 10%	*DB*:*NaturalPlace* (*D*10) 10%
Selected properties	*team*, *name*, *position*, *birthPlace*, *datedeNaissance*, *squadNumber*, *currentMember*	*team*, *name*, *position*, *birthPlace*, *datedeNaissance*, *squadNumber*, *currentMember*, *occupation*, *dernierePubli*, *birthYear*	*lat*, *long*, *name*, *point*, *country*, *district*, *location*, *elevation*, *rivermouth*, *locatedinarea*, *sourceCountry*

Table 3.3 Selected properties for the classes of DBpedia

In order to show the scalability of our method applying the optimistic heuristic, we have applied KD2R on two datasets extracted from DBpedia[7]: the first dataset contains descriptions of persons and the second one of natural places (see Table 3.1). One of the characteristics of DBpedia is that the UNA is not fulfilled. All the keys that can be discovered on such a dataset would still remain valid even if the duplicates are removed. However, some of the possible minimal keys can be lost. In DBpedia, we observe that some people are represented several times using distinct instances, but in different contexts. For example, one soccer-player is represented using several instances, but for each instance the description concerns its transfer into an new club. In such cases, keys can be discovered.

On small datasets such as OAEI datasets or GFT (less than 10 000 triples), KD2R can be applied using both the pessimistic and the optimistic heuristics. Nevertheless, on large datasets such as DBpedia persons (more than 5.6 million of triples) or DBpedia natural places (more than 1.6 million of triples), the approach based on the pessimistic heuristic cannot be used. Indeed, such datasets contain a lot of properties that are rarely instantiated which leads to a *FP-Tree* that contains too many nodes due to the assignation of all the possible values to the artificial "null" values in the prefix tree. Hence, in such cases only the optimistic heuristic can be applied. Moreover, in our experiments we have considered only the properties that are instantiated for at least T distinct instances of *DB*:*Person* and *DB*:*NaturalPlace* since KD2R cannot scale when the number of properties is big. Table 3.3 depicts the selected properties of the classes of DBpedia.

[7] http://dbpedia.org/Downloads37

The first dataset $D9$ contains 763644 instances of the class *DB*:*Person* which corresponds to 5639680 RDF triples. The second dataset $D10$ contains 49887 instances of the class *DB*:*NaturalPlace*, or 1604347 RDF triples. To show how the inherited keys are exploited, KD2R has been applied on the class *DB*:*NaturalPlace*, its subclass *DB*:*BodyOfWater* and on the class *DB*:*Lake* which is a subclass of *DB*:*BodyOfWater*. For the class *DB*:*Person* of $D9$, when T is equal to 20%, the set of obtained keys is given in Table 3.4. When T is equal to 10%, KD2R obtains 17 additional composite keys, such as $\{name, occupation, birthdate, activeyearstartyear, birthplace\}\}$ and $\{name, position, deathdate\}$. For the class *DB*:*NaturalPlace* of $D10$, when T is equal to 20%, the set of obtained keys is given in the table Table 3.5.

***DB:Person* (T=20%)**
$\{\{squadnumber, birthplace\}$
$\{squadnumber, birthdate\}$
$\{currentmember, birthplace\}$
$\{currentmember, name\}$
$\{squadnumber, name\}$
$\{currentmember, birthdate\}\}$

Table 3.4 Keys for the class *DB*:*Person*.

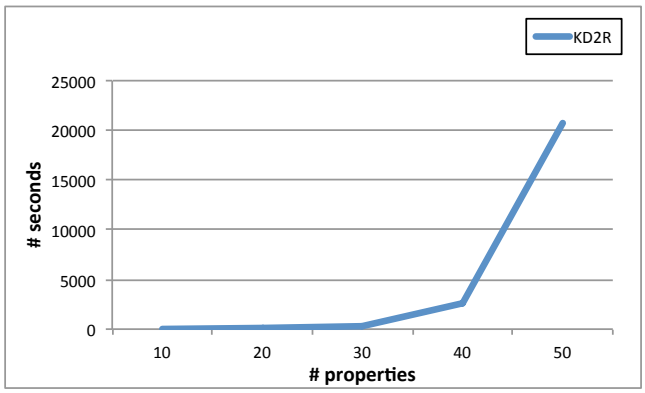


Fig. 3.13 UNKeyFinder runtime for the class *DB*:*NaturalPlace*

For the 33993 instances of the class *DB*:*BodyOfWater*, we have found 13 keys, four of which are subsets of some minimal keys that are inherited from *DB*:*NaturalPlace* like $\{lat, district\}$. The rest of the minimal keys belong to the set of minimal keys inherited from *DB*:*NaturalPlace*.

***DB:NaturalPlace* (T=20%)**
{{*name*, *district*, *elevation*},
{*sourcecountry*, *location*},
{*country*, *district*, *long*},
{*district*, *sourcecountry*, *elevation*},
{*sourcecountry*, *long*},
{*district*, *location*},
{*name*, *lat*, *district*},
{*country*, *locatedinarea*},
{*lat*, *district*, *elevation*},
{*lat*, *sourcecountry*},
{*location*, *locatedinarea*},
{*sourcecountry*,*locatedinarea*},
{*district*, *locatedinarea*},
{*name*,*district*, *point*},
{*country*, *lat*, *district*},
{*name*, *district*, *long*},
{*district*, *elevation*, *long*},
{*country*, *sourcecountry*, *elevation*},
{*country*, *district*, *point*},
{*district*, *point*, *elevation*},
{*sourcecountry*, *point*}}

Table 3.5 Keys for the class *DB*:*NaturalPlace*.

For the 9438 instances of the class *DB*:*Lake*, we have found 7 minimal keys, three of them are subsets of some minimal keys that are inherited from *DB*:*BodyOfWater* like {*sourceCountry*}. The other minimal keys belong to the set of minimal keys inherited from *DB*:*BodyOfWater*.

To measure the resistance of KD2R to the size of properties, we have run KD2R on sets of properties of different sizes. In Figure 3.13, we observe that the time increases exponentially to the size of properties and after a certain number the algorithm runs out of memory.

3.5.1.5 KD2R results on INA and ABES datasets

In the context of the Qualinca project, two additional datasets have been tested by KD2R. Qualinca ("*Qualité et interopérabilité de grands catalogues documentaires*") is a research project funded by the National Research Agency of France (ANR). The objective of this project is to develop mechanisms allowing to quantify the quality level of a bibliographical

knowledge base. Moreover, the improvement of the mentioned quality level, the maintenance of the quality when updating the knowledge base and finally, the use of the knowledge bases taking into account their quality level are also very important goals of this project. The project started in April 2012 and ends in March 2016.

The first dataset $D1$1 comes from the *"QNational Audiovisual Institute"* (INA) of France while the second $D1$2, contains data from the *Bibliographic Agency for Higher Education* (ABES). Both datasets, contain metadata extracted from large catalogs.

INA dataset. The dataset coming from INA contains RDF descriptions of audiovisual contents (class *Contenu*) and people that are involved in these contents (class *Personne*). More precisely, 44779 instances of the class *Contenu* are described by 82 properties (25 single valued properties, 57 multivalued properties) and 7444 instances of the class *Personne* are described by 44 properties (21 single valued properties and 23 multivalued properties). A class named *PrecisionNode* is used to link contents to persons. Many properties are used to describe only a few number of instances (52 properties appear in less than 1% of the *Contenu* instances, and 30 of the properties appear in less than 1% of the *Personne* instances). Thus, we set T at 30% in order to discover keys that are meaningful and applicable to a large number of instances. A pre-selection of the data is also necessary since KD2R cannot scale when the data contain the whole set of properties.

For the class *Contenu*, the selected properties are the following: *ina*:*aPourTitrePropreIntegrale*, *ina*:*aPourDateDiffusion*, *ina*:*aPourFonds*, *ina*:*aPourTypeNotice*, *ina*:*aPourTitreCollection*, *ina*:*aPourGenre*, *ina*:*aPourDuree*, *ina*:*aPourTheme*, *ina*:*aPourDateCreationNotice*, *ina*:*aPourParticipant*.

The property *ina*:*aPourTitreCollection* represents the general title of a show while *ina*:*aPourTitrePropreIntegrale* the title of a particular episode. The property *ina*:*aPourGenre* contains the category of a content, the properties *ina*:*aPourDuree* and *ina*:*aPourDateDiffusion*, the duration and the date that this content took place while *ina*:*aPourTheme* the subject of the show. The property *ina*:*aPourDateCreationNotice* corresponds to the date where this description was added to the database. Moreover, the property *ina*:*aPourParticipant* describes the people that participated in this content while the property *ina*:*aPourFonds* the corpus from where the information come from. Finally, the property *ina*:*aPourTypeNotice* groups the contents in categories such as extract or tv show.

When setting T at 30%, the number of treated triples for the class *Contenu* is 470452. KD2R has found no keys for the class *Contenu* since the UNA is not fulfilled in this part of the dataset. Indeed, as shown in Table 3.6, there exist two descriptions sharing values for

Properties	Instance LM00001239918	Instance LM00001239919
ina:*PourTitrePropreIntegrale*	Soirée élections régionales et cantonales 1er tour	Soirée élections régionales et cantonales 1er tour
ina:*aPourGenre*	Débat	Débat
ina:*aPourTheme*	concept10236135	concept10236135
ina:*aPourDuree*	2400	2400
ina:*aPourTypeNotice*	Notice sujet	Notice sujet
ina:*aPourDateCreationNotice*	3/16/98	3/16/98
ina:*aPourParticipant*	Moulinard Christian	Moulinard Christian
ina:*aPourFonds*	Actualités	Actualités
ina:*aPourDateDiffusion*	3/15/98	3/15/98
ina:*aPourTitreCollection*	Spécial élections	Spécial élections

Table 3.6 Duplicate instances for the class *Contenu*

every selected property.

With the same threshold, the number of treated triples for the class *Personne* is 125963. For the class *Personne*, the selected properties are the following: *ina*:*prefLabel*, *ina*:*hiddenLabel*, *ina*:*aPourNoteQualite*, *ina*:*aPourStatut*, *inv-ina*:*aPourParticipant*, *inv-ina*:*aPourConcept*, *ina*:*aPourSexe*, *inv-ina*:*imageContient*.

The properties *ina*:*prefLabel* and *ina*:*hiddenLabel* represent both the name of a person, in lowercase and uppercase respectively, while the property *ina*:*aPourSexe* represents the gender of a person. The property *ina*:*aPourNoteQualite* describes the status of a person related to the contents. For example, journalist, camera man etc. The origin of a value is given in the property *ina*:*aPourStatut*. For example, a value can be found in a thesaurus or a dictionary. The properties *inv-ina*:*aPourParticipant*, *inv-ina*:*aPourConcept* and *inv-ina*:*imageContient* represent the inverse properties of the properties *ina*:*aPourParticipant*, *ina*:*aPourConcept* and *ina*:*imageContient* of the class *Contenu*.

Applying the optimistic heuristic, KD2R discovers three keys:

$K_{D11.Personne}$= {{*inv-ina*:*aPourConcept*}, {*ina*:*HiddenLabel*}, {*ina*:*prefLabel*}}

These keys have been validated by INA experts but they are not considered as useful for an interlinking task or a task that aims to validate existing links. Indeed, by construction, the label that describes a person is created by an INA member for each new person. Thus, this property has, by default, distinct values. In the same way, the property *inv-*

ina:*aPourConcept* links *PrecisionNode*s to persons and is a functional property by construction.

Applying the pessimistic heuristic to this class, KD2R discovers two keys:
$K_{D11.Personne}$= {{*ina*:*HiddenLabel*}, {*ina*:*prefLabel*}}.

ABES dataset. The ontology FRBR00, coming from the ABES dataset, contains a large amount of bibliographic notices. The data contain more than 11 million of bibliographic notices and 2.5 million of authority notices.

KD2R has been applied on two different datasets containing subparts of the data provided by ABES. The first extract of ABES, $D12$, contains a set of authority notices that represent 5671 persons, described by 9 properties, and contains 59839 RDF triples. The properties used to describe a person are the domain of a publication *domainPubli*, the date of the first publication *premierePubli*, the date of the last publication *dernierePubli*, the language in which the publication is written *langue*, the author's first name *apourprenom*, last name *apournom*, first name and last name *apourappellation*, date of birth *datedeNaissance* and finally his death date *dernierePubli*, if it exists. These notices are obtained from an original dataset conforming to the FRBR00 ontology.

Since KD2R cannot discover keys involving chains of properties, the dataset has been transformed, by another partner of the project, LIRMM (Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier), to a compact representation where a chain of properties is transformed into a single property. The keys found by KD2R applying the optimistic heuristic are presented in Table 3.7. The pessimistic heuristic of KD2R cannot be applied in this dataset.

The second extract $D13$, contains 573 descriptions of contextual entities, generated using the bibliographic notices that conform to the ontology FRBR00. This extract contains 8738 triples. 13 different properties are used to describe these contextual entities. For example, a contextual entity that represents an instance of a person, contains a person's first name, last name and the domain of a bibliographic notice correlated to that person such as computer science or biology. Moreover, the data are enriched with other information such as the role of a person that can be, for example author or editor.

The keys discovered by KD2R in $D13$, are presented in Table 3.8. An ABES expert has evaluated the discovered keys. The value 0 is assigned to sets of properties that are not actual keys, the value 1 to sets of properties that contain usually few exceptions and the value 2 to real keys. Finally, the symbol ? is used to state that the expert cannot distinguish

{{*domainPubli*, *premierePubli*, *dernierePubli*, *apournom*, *datedeNaissance*},
{*dernierePubli*, *apourprenom*, *datedeNaissance*, *dernierePubli*},
{*premierePubli*, *apourprenom*, *datedeNaissance*, *dernierePubli*},
{*domainPubli*, *dernierePubli*, *datedeNaissance*, *dernierePubli*},
{*domainPubli*, *premierePubli*, *apourprenom*, *dernierePubli*},
{*dernierePubli*, *apourprenom*, *apournom*, *dernierePubli*},
{*domainPubli*, *dernierePubli*, *apournom*, *dernierePubli*},
{*premierePubli*, *apourappellation*, *datedeNaissance*},
{*premierePubli*, *apourappellation*, *dernierePubli*},
{*dernierePubli*,*apourappellation*, *dernierePubli*},
{*premierePubli*, *dernierePubli*, *dernierePubli*},
{*premierePubli*, *apournom*, *dernierePubli*}}.

Table 3.7 Keys found by KD2R in $D12$ applying the optimistic heuristic

{*apourcollectivite*, *apourprenom*, *apourcoauteur*},	0
{*apourcollectivite*, *apournom*, *apourdatepubli*},	1
{*apourdomaine*, *apourprenom*, *apourcoauteur*},	0
{*apourcollectivite*, *apourdomaine*, *apournom*},	1
{*apourcollectivite*, *apourprenom*, *apournom*},	0
{*apourrole*, *apourprenom*, *apourcoauteur*},	0
{*apourcollectivite*, *apourrole*, *apournom*},	0
{*apourcollectivite*, *apourappellation*},	0
{{*apourcoauteur*, *apourappellation*},	1
{*apourcoauteur*, *apourdatepubli*},	0
{*apourappellation*, *apourtitre*},	1
{*apourcollectivite*, *apourtitre*},	0
{*apourmanifestationassociee*},	0
{*apourcoauteur*, *apourtitre*},	0
{*apourcoauteur*, *apournom*},	1
{*apourprenom*, *apourtitre*},	0
{*apournom*, *apourtitre*},	0
{*apourautoriteassociee*}}.	?

Table 3.8 Keys found by KD2R in $D13$ applying the optimistic heuristic

between the cases. Notice that none of the discovered keys is considered as a real key by the expert while in most of the cases these sets are considered as keys with exceptions or not actual keys.

dataset	pruning category	#not-visited-nodes	not-visited rate	#nodes without pruning	time with pruning (s)	time without pruning (s)
OAEI:*Person*1 - *D*1	Monotonicity	764478	60%	1252994	4	8
OAEI:*Person*2 - *D*2	Monotonicity	1679956	75%	2234738	8	10
OAEI:*Restaurant*1 - *D*3	Monotonicity	228	81%	280	1	2
OAEI:*Restaurant*2 - *D*4	Monotonicity	103	71%	146	1	2
GFT - *D*5	Monotonicity	84	10%	827	1	3
ChefMoz - *D*6	Monotonicity	71754	55%	129569	570	625

Table 3.9 Pessimistic heuristic: search space pruning and runtime results

3.5.2 Scalability of KD2R

The complexity of the prefix tree exploration is exponential in terms of the number of the property values. In order to test the scalability of our method we have checked experimentally on the seven datasets the benefits of the different kinds of pruning that are used during the prefix tree exploration. More specifically, as it is already mentioned, KD2R exploits three following pruning strategies (see Section 3.3.3.1): the key pruning, the anti-monotonic pruning and the mononotonic pruning.

Table 3.9 shows the results of KD2R in terms of runtime and search space pruning for every dataset when the pessimistic heuristic is applied. Similarly, Table 3.10 presents the results when the optimistic heuristic is used. In both tables, the results correspond to the sum of obtained results for each class in the dataset. For example, the results given for the dataset *D*1 represent the results for both *Person* and *Address* classes.

The pruning strategies enable KD2R to be more efficient and scalable in big datasets. Both Tables 3.9 and 3.10 show that on the five smallest datasets, the execution time of keyFinder (using pessimistic or optimistic) is less than 8 seconds. For the two DBpedia datasets, the execution time is less than 441 seconds when a subset of the properties is selected. Thanks to different kinds of pruning presented in Section 3.3.3.1, less than 50% of the nodes of the prefix tree are explored for all datasets. It should be mentioned that for the instances of the class DBpedia *DB*:*Person* and the class DBpedia *DB*:*NaturalPlace*, less than 5% and 1% of the nodes are explored respectively. The sumbsumption relations between the classes of the dataset *D*5 are used to show the importance of the key pruning strategy. 13% of all the prunings that take place in this dataset are obtained thanks to the key pruning (See Table 3.10).

Nevertheless, even if the prunings clearly improve the execution time, KD2R cannot deal with big datasets containing a big number of properties.

dataset	pruning category	# not visited nodes	not visited rate	# nodes without pruning	time with prunings	time without prunings
OAEI:*Person*1 - *D*1	Monotonicity	12156	88%	13750	3	7
OAEI:*Person*2 - *D*2	Monotonicity	16225	89%	18276	3	5
OAEI:*Restaurant*1 - *D*3	Monotonicity	228	81%	280	1	2
OAEI:*Restaurant*2 - *D*4	Monotonicity	103	71%	146	1	2
GFT - *D*5	Monotonicity	108	22%	499	1	3
ChefMoz - *D*6	Monotonicity	27026	55%	49351	5	8
DB:*Person* - *D*9 (T=20%)	Monotonicity	27302986	95%	28803153	441	634
DB:*NaturalPlace* - *D*10 (T=20%)	Monotonicity	40907348	99%	47716771	42	222
	Antimonotonicity	159538				
	Key Inheritance	6153252				
OAEI:*Film*1 - *D*7	Monotonicity	223436	99 %	12836301	30	75
	Antimonotonicity	10492012				
OAEI:*Film*2 - *D*8	Monotonicity	2313411	98%	9135607	40	82
	Antimonotonicity	6701456				

Table 3.10 Optimistic heuristic: search space pruning and runtime results

3.5.3 Data linking with keys

To evaluate the quality of the discovered keys we use them to link data. We consider datasets where the complete set of links between them, i.e., gold standard, is available. Once the data are linked, we compare the linked instances to the correct links existing in the gold standard. The recall, precision and F-measure are used to evaluate the quality of the results.

The recall in our case corresponds to the ratio of retrieved links that are relevant to the total number of relevant links.

$$\text{recall} = \frac{|\{\text{relevant links}\} \cap \{\text{retrieved links}\}|}{|\{\text{relevant links}\}|}$$

The precision corresponds to the ratio of retrieved links that are relevant to the total number of retrieved links.

$$\text{precision} = \frac{|\{\text{relevant links}\} \cap \{\text{retrieved links}\}|}{|\{\text{retrieved links}\}|}$$

Finally, the F-measure is the harmonic mean of precision and recall.

$$\text{F-measure} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

As it will be shown in the following sections, in the first experiment (see Section 3.5.3.1)

no similarity measures are used to compare the values of keys. Two values are considered as the same only when they are strictly equal. Additionally, no decision propagation is used. In the second experiment presented in Section 3.5.3.2, the recall, precision and F-measure of the results are computed when the keys are applied all together using similarity measures between values and where the linking decisions are propagated thanks to N2R tool (see section 3.5.3.2).

3.5.3.1 Key quality without similarity measures and without propagation

In this experiment, we compute the recall, the precision and the F-measure obtained on datasets $OAEI{:}Film1$ and $OAEI{:}Film2$ when keys are applied using equality between values (instead of using similarity scores). Moreover, linking decisions are not propagated (e.g. "*same restaurants, then same addresses, then same cities, then, ...*").

Due to the data heterogeneity of the datasets, we have obtained a low value for the recall (5.03%). Discovering keys that are valid for both datasets allows to guarantee a very high value for the precision. This is shown by the obtained precision which is of 100%. This leads to an F-measure of about 10%. These results show that the obtained keys have a good quality in terms of correctness. However, due to the heterogeneity of the data and to the fact that the decision propagation is not applied, the recall is very low. Hence, in order to ensure the good quality results, in terms of recall and precision, more complex linking tools that take similarity measures into account become necessary.

3.5.3.2 Key quality by using similarity measures and decision propagation

In this section, we evaluate the quality of keys when they participate in more complex data linking rules. An real data linking tool has been used to show the benefits of using discovered keys in the data linking process. This experiment consists of three different scenarios. In the first scenario, the *owl:sameAs* links are computed considering that no keys are available. As a second scenario, the links are computed using keys manually defined by an expert in the ontology, for the OAEI 2010 contest. Finally, in the third scenario, the keys are discovered by KD2R and declared in the ontology. We compute the precision, recall and F-measure for the three scenarios and we compare the results.

Brief presentation of N2R. N2R is a knowledge based approach exploiting the keys declared in the ontology to infer *owl:sameAs* links between class instances. It exploits keys in order to generate a function that computes similarity scores for pairs of instances. This

numerical approach is based on equations that model the influence between similarities. In the equations, each variable represents the (unknown) similarity between two instances while the similarities between values of datatype 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 considered in the equation system. More precisely, for each reference pair, the similarity score is modeled by a variable x_i and the way it depends on other similarity scores is modeled by an equation: $x_i = f_i(X)$. In this equation $i \in [1..n]$, n is the number of reference pairs for which we apply N2R, and $X = (x_1, x_2, \ldots, x_n)$ is the set of their corresponding variables. Each equation $x_i = f_i(X)$ is of the form:

$$f_i(X) = max(f_{i-df}(X), f_{i-ndf}(X))$$

The function $f_{i-df}(X)$ represents the maximum similarity score obtained for the instances of the datatype properties and the object properties that belong to a key describing the i-th reference pair. In case of a composite key we compute first the average of the similarity scores of the property instances involved in that combined key. The maximum function allows to propagate the similarity scores of the values and the instances having a strong impact. The function $f_{i-ndf}(X)$ is defined by a weighted average of the similarity scores of the literal value pairs (and sets) and the instance pairs (and sets) of datatype properties and object properties describing the i-th instance pair and not belonging to a key. See [SPR09] for the detailed definition of $f_{i-df}(X)$ and $f_{i-ndf}(X)$. Solving this equation system is performed by an iterative method inspired by the Jacobi method [GL96], which is quickly converging on linear equation systems.

The instance pairs for which the similarity is greater than a given threshold $TRec$ are linked, i.e, an *owl:sameAs* link is created between the two instances.

Obtained results on OAEI 2010 datasets. Tables 3.11 and 3.12 show the results obtained by N2R in terms of recall, precision and F-measure when: (i) no keys are used, (ii) all KD2R keys are used and (iii) keys defined by experts are used (details can be found in [SNPR10]). Since the domains concerning persons and restaurants are rather common, the expert keys have been declared manually by one of the participants of the OAEI 2010, for N2R tool. If several experts are involved, a *kappa* coefficient [Coh68] can be computed to measure their agreement. Since $D1$ contains properties that are not instantiated, both the optimistic and pessimistic heuristics have been performed. In Table 3.11, we define as KD2R-O the

results obtained using keys discovered with the optimistic heuristic and KD2R-P the results obtained using keys discovered with the pessimistic heuristic. It should be mentioned that for the datasets $D2$ and $D3$ the results given for KD2R are both the results of KD2R-O and KD2R-P, since there are no undetermined keys. We show the results when the threshold $TRec$ varies from 1 to 0.8. Since the F-measure expresses the trade-off between the recall and the precision, we first discuss the obtained results according to this measure. Across all datasets and values of $TRec$, the F-measure obtained using KD2R keys is greater than the F-measure obtained when keys are not available. We can notice that, the results obtained for the Person dataset $D1$ are better when we use keys obtained by either KD2R-O or KD2R-P than when the keys are not used. When the threshold is bigger than 0.95 the F-measure of N2R using KD2R-O keys is 100%. This is an example that shows that the results using keys found with the optimistic heuristic can be better than the ones found with the pessimistic heuristic. In the restaurant dataset $D2$, when $TRec \geq 0.9$, the F-measure is almost three times higher than the F-measure obtained when keys are not declared. This big difference is due to the fact that the recall is much higher when KD2R keys are added. Indeed, even when some property values are syntactically different, it suffices that it exists one key for which the property values are similar, to infer any identity link. For example, when $TRec = 1$, the KD2R recall is 95% for the persons dataset while without the keys the recall is 0%. Hence, the more numerous the keys are, the more identity links can be inferred.

Furthermore, our results are very close to the ones obtained applying expert keys. For both datasets, the largest difference between KD2R F-measure and the expert's one is 6%. We should also mention that KD2R precision is always higher than the expert precision. Indeed, some expert keys are not verified in the dataset. For example, while the expert has declared *phoneNumber* as a key for the *Restaurant* class, in this dataset some restaurants share the same phone number, i.e., they are managed by the same organization.

These results show that the data linking results are significantly improved, especially in terms of recall, when we compare them to results that can be obtained when the keys are not defined.

In Table 3.13, we give a comparison between the results obtained by N2R using KD2R keys, with other tools that have used the datasets *OAEI*:*Person*1, *OAEI*:*Person*1, *OAEI*:*Restaurant*1, *OAEI*:*Restaurant*2 of OAEI 2010–Instance Matching track. We notice that the obtained results, in terms of F-measure, are comparable to those obtained by semi-supervised approaches like ObjectCoref [HCQ11]. It is, however, less efficient than approaches that learn linking rules that are specific to the dataset like KoFuss+GA.

TRec	Keys	Recall	Precision	F-measure
1	without	0%	- %	- %
	KD2R-O	100%	100%	100%
	KD2R-P	95.00%	100%	97.44%
	expert	98.40%	100%	99.19%
0.95	without	61.20%	100%	75.93%
	KD2R-O	100%	100%	100%
	KD2R-P	95.00%	100%	97.44%
	expert	98.60%	100%	99.30%
0.9	without	64.2%	100%	78.20%
	KD2R-O	100%	98.04%	99.01%
	KD2R-P	95.00%	100%	97.44%
	expert	98.60%	100%	99.30%
0.85	without	65.20%	100%	78.93%
	KD2R-O	100%	81.30%	89.68%
	KD2R-P	99.80%	100%	99.90%
	expert	99.80%	100%	99.90%
0.8	without	90.20%	100%	94.85%
	KD2R-O	100%	35.71%	52.63%
	KD2R-P	99.80%	100%	99.90%
	expert	100%	100%	100%

Table 3.11 Recall, Precision and F-measure of data linking for $D1$ and $D2$

Obtained results for GFT-ChefMoz dataset. Table 3.14 shows the results obtained by N2R in terms of recall, precision and F-measure when: (i) no keys are used and (ii) KD2R keys are used. We show the results when the threshold $TRec$ takes values in the interval $[0.7, 1]$. For both datasets and for every $TRec$ value, the F-measure found using KD2R keys is greater than the F-measure when keys are missing.

This difference is due to the fact that the recall is always higher when KD2R keys are added. Indeed, even when some property values are syntactically different, it suffices that there exists one key for which the property values are similar, to infer the identity links. For example, when $TRec = 1$, the KD2R recall is 60% for the persons dataset while without the keys the recall is 45%. Hence, the more numerous the keys are, the more identity links can be inferred.

As in $D1$ and $D2$, the above results show that the data linking results are significantly improved, in particular in terms of recall, when we compare them to results obtained when the keys are not defined.

TRec	Keys	Recall	Precision	F-measure
1	without	0%	- %	- %
	KD2R	62.50%	80.46%	70.35%
	expert	76.79%	74.78%	75.77%
0.95	without	14.29%	80.00%	24.24%
	KD2R	62.50%	80.46%	70.35%
	expert	77.68%	75.00%	76.32%
0.9	without	14.29%	80.00%	24.24%
	KD2R	62.50%	80.46%	70.35%
	expert	77.68%	75.00%	76.32%
0.85	without	14.29%	80.00%	24.24%
	KD2R	65.17%	80.22%	71.92%
	expert	77.68%	75.00%	76.32%
0.8	without	37.5%	80.76%	51.21%
	KD2R	66.96%	79.78%	72.81%
	expert	77.68%	75.00%	76.32%

Table 3.12 Recall, Precision and F-measure of data linking for $D3$ and $D4$

Dataset	N2R + KD2R-P	N2R + KD2R-O	ASMOV	N2R	CODI	ObjectCoref	RIMOM	KnoFuss + GA
$D1$-$D2$	0.99	1.00	1.00	1.00	0.91	1.00	1.00	1.00
$D3$-$D4$	0.728	–	0.70	0.75	0.72	0.73	0.81	0.78

Table 3.13 Comparison of data linking F-measure with other tools on person datasets $D1$ and $D2$ of OAEI 2010 benchmark

3.5.3.3 KD2R keys vs. SF keys

Different definitions of keys can be considered in the Semantic Web. Unlike KD2R where two instances are the same when they share at least one value for each property of a key, in [ADS12] they should share all the values (see Section 2.2.1). In the following, to distinguish the two types of keys we refer to the keys discovered by KD2R as KD2R keys and the keys discovered by [ADS12] as SF keys.

Quantative evaluation. To compare these approaches, we discover both KD2R keys and SF keys in the dataset $D7$ and they are used to link the datasets $D7$ and $D8$. To link the datasets a string equality after stop words elimination for datatype properties and equality for object properties have been applied.

TRec	Keys	Recall	Precision	F-measure
1	without	45.67%	100%	62.71%
	KD2R	60.49%	100%	75.38%
0.95	without	50.61%	100%	67.21%
	KD2R	60.49%	100%	75.38%
0.9	without	50.61%	100%	67.21%
	KD2R	60.49%	100%	75.38%
0.85	without	50.61%	100%	67.21%
	KD2R	60.49%	100%	75.38%
0.8	without	54.32%	100%	70.39%
	KD2R	60.49%	100%	75.38%
0.75	without	54.32%	100%	70.39%
	KD2R	60.49%	100%	75.38%
0.7	without	60.49%	100%	75.38%
	KD2R	61.72%	100%	76.33%

Table 3.14 Recall, Precision and F-measure for $D5$ and $D6$

We remind that the KD2R keys discovered in the dataset $D7$ (see Section 3.5.1.3) are: $\{\{name, directedBy, filmedIn\}, \{article\}, \{estimatedBudget\}\}$.

The SF keys that are discovered in the dataset $D7$ are: $\{\{name, directedBy, filmedIn\}, \{article\}, \{estimatedBudget\}, \{name, featuring\}, \{name, starringIn\}\}$.

To evaluate the quality of the discovered keys, they have been applied to compute identity links between film instances of the datasets $D7$ and $D8$. More precisely, we have measured the quality of each set of keys independently from the quality of the possible links that can be found for other class instances. Having this goal, we have exploited the correct links appearing in the gold standard to compare object property values. In table 3.15, we present recall, precision and F-measure for each type of keys.

keys	Recall	Precision	F-Measure
KD2R keys	27.86%	100%	43.57%
SF keys	27.86%	100%	43.57%

Table 3.15 Recall, Precision and F-measure for the class $Film$

We notice that the results of KD2R keys and SF keys are the same. Indeed, all the

links that are found by the two additional SF keys are included in the set of links obtained using the three shared keys. Furthermore, the shared keys generate the same links either because the involved properties are mono-valued (*estimatedBudget*, *article*), or because the involved multi-valued object properties have the same values in both files for the same film.

Some SF keys cannot be KD2R keys and may have a higher recall. For example, it is not sufficient to know that two films share one main actor to link them. On the other hand, when the whole sets of actors are the same, two films can be linked with a good precision. Moreover, if we consider only instances having at least two values for the property *starringIn* (98% of the instances), this property is discovered as an SF key and allows to find links with a precision of 96.3% and a recall of 97.7%.

Growth resistance evaluation. We have evaluated how the quality of each type of keys evolve when they are discovered in smaller parts of the dataset. Thus, we have randomly split the dataset *D*7 in four parts. Each part contains the complete description of a subset of the *Film* instances. We have then discovered the keys in a file that contains only 25% of the data, 50% of the data,and finally 75% of the data. Then we have computed the recall, precision and F-measure that are obtained for each type of keys. The larger the dataset, the more specific are the keys. Also, for all types of keys, precision increases and recall decreases with dataset' size. To obtain a good precision, KD2R keys need to be discovered in at least 50% of this dataset, while SF keys obtain a rather good precision when the keys are learnt using only 25% of the dataset. Furthermore, some SF keys have also a very high recall when they are learnt on a subpart of the dataset even if the precision is not 100%. Indeed, new RDF descriptions are introduced that prevent the system from discovering keys that can be very relevant.

KD2R keys	Recall	Precision	F-Measure
25%	27.85%	77.55%	40.98%
50%	27.85%	99.42%	43.51%
75%	27.85%	99.42%	43.51%
100%	27.85%	100%	43.56%

SF keys	Recall	Precision	F-Measure
25%	100%	94.1%	96.96%
50%	100%	99.03%	99.51%
75%	27.85%	99.42%	43.51%
100%	27.85%	100%	43.56%

Table 3.16 Recall, Precision and F-measure for KD2R keys and SF keys

3.6 Conclusion

In this chapter, we have presented KD2R, an approach that discovers keys in RDF datasets. KD2R can tackle datasets where UNA is known. To obtain keys that are valid for different datasets that may conform to distinct ontologies, we discover keys containing aligned properties found in every explored dataset. Once all the keys are discovered, we apply a merge step to find the set of minimal keys that are valid in every dataset.

KD2R takes into account characteristics of RDF data such as incompleteness and multivaluation. KD2R proposes two different heuristics in order to work with incomplete data, the pessimistic heuristic and the optimist heuristic. Since the data may be numerous, a strategy that discovers first maximal non keys that are used to compute keys is adopted. Indeed, to discover that a set of properties is a non keys only a subpart of the data are required.

KD2R applies different pruning strategies, some of which were initially introduced in the setting of relational databases. A novel pruning that exploits the key inheritance to prune the key search for a given class is also proposed. This pruning is only applicable in the setting of the Semantic Web.

The experiments have been conducted on thirteen datasets. Among them, six datasets have been used in the instance matching track of OAEI evaluation initiative, four datasets have been collected from the Web of data and finally, three datasets have been tested in the context of the Qualinca project. The experiments showed that KD2R can perform well in small datasets where the number of properties found in the data is limited. When large datasets, containing many properties, are applied, KD2R cannot scale. In every case, the optimistic heuristic is faster than the pessimistic heuristic. However, the derivation of keys from non keys remains the bottleneck of KD2R.

To evaluate the linking power of the discovered keys, we have linked data using (i) KD2R keys (ii) expert keys and (iii) no keys. The experiments have shown that when KD2R keys are used, the data linking results are better than applying no keys and similar to the ones using expert keys. Comparing keys found with the optimistic heuristic and the pessimistic heuristic, the optimistic keys lead to better linking results in the tested datasets. Finally, the experiment conducted to compare KD2R keys with SF keys has not led to significant conclusions due to the particular characteristics of the selected dataset.

Chapter 4

SAKey: Scalable Almost Key discovery in RDF data

Over the last years, the Web of data has received a tremendous increase, containing a huge number of RDF triples. Data published on the Web are usually created automatically, therefore they may contain erroneous information. Moreover, distinct URIs that refer to the same real world object, i.e., duplicates, may exist in the data. Considering that a key uniquely identifies every instance in a dataset, if data containing erroneous information or duplicates are used in the key discovery, relevant keys can be lost. Thus, algorithms that search only for keys, are not able to discover all the keys in such datasets. For this reason, it becomes essential to develop approaches that allow the discovery of keys despite the presence of some instances that violate them. Instances that lead to these violations are called exceptions.

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 few exceptions can prevent the loss of keys. It is important to mention that for approaches like N2R [SPR09] that are based on keys to link data, the more keys they take into account, the more significant results they can produce. Furthermore, sets of properties that are not exact keys due to few exceptions can lead to many identity links with a reasonable error rate. For example, the *telephone number* of a restaurant can be used as a key in data linking process, even if there may exist few restaurants located in the same place sharing phone numbers. In this case, even if this property is not a real key, it can be useful in the linking process.

An important characteristic of RDF datasets that are available on the Web is their big volume. To deal with this, we have first developed KD2R (see Chapter 3), that discovers first the complete set of maximal non keys and then derives the set of minimal keys from them.

This makes the discovery of keys more efficient. Nevertheless, as shown in the Section 3.5, KD2R is overwhelmed by the huge amount of data found on the Web. To improve the scalability of the key discovery approach we present in this chapter a new method called SAKey (Scalable Almost Key discovery), that is able to discover keys on big datasets in the presence of errors and/or duplicates. We call the keys discovered by SAKey, *almost keys*. An almost key represents a set of properties that is not a key due to few exceptions. As in KD2R, 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 filtering techniques and pruning strategies that reduce the requirements of time and space of the non key discovery. Since the derivation of keys from non keys is considered as the bottleneck of KD2R, SAKey introduces a new efficient key derivation algorithm. Finally, an extension of SAKey for the discovery of *conditional keys*, i.e., keys that are valid under a specific condition, is proposed.

To deal with the incompleteness of data, SAKey considers that every value missing from the data is different from what exists in the data. This assumption corresponds to the optimistic heuristic, first introduced by KD2R, that has been shown (see Section 3.5) to be much faster and leading to better data linking results than the pessimistic heuristic.

The work described in this chapter appears in [SAPS14].

In what follows we present our contributions concerning SAKey approach. The main contributions are:

1. The introduction of a heuristic to discover keys, that prevents us from losing keys in data that contain erroneous information and/or duplicates.

2. An algorithm that efficiently discovers *non keys* applying a series of filtering steps and pruning strategies.

3. A new algorithm for an efficient derivation of almost keys from non keys which until now was considered as the bottleneck of the approaches that discover non keys first.

This Chapter is organized as follows. Section 4.1 contains an example that will be used throughout this chapter. Section 4.2 formalizes the problem we consider. Section 4.3 is the main part of the Chapter, presenting the almost keys and their discovery using SAKey. Section 4.4 explains the computation of valid almost keys coming from different datasets while Section 4.5 introduces C-SAKey, an extension of SAKey for the discovery of conditional keys. Finally, Section 4.6 describes our experiments before Section 4.7 concludes.

o1:Film(f10), $o1{:}director(f10,''O.Nakache'')$, $o1{:}director(f10,''E.Toledano'')$, $o1{:}hasActor(f10,''F.Cluzet'')$, $o1{:}hasActor(f10,''O.Sy'')$, $o1{:}releaseDate(f10,''2/11/11'')$, $o1{:}name(f10,''The\ Intouchables'')$,

o1:Film(f11), $o1{:}director(f11,''O.Nakache'')$, $o1{:}director(f11,''E.Toledano'')$, $o1{:}hasActor(f11,''F.Cluzet'')$, $o1{:}hasActor(f11,''O.Sy'')$, $o1{:}releaseDate(f11,''2/11/11'')$, $o1{:}name(f11,''The\ Intouchables'')$,

Fig. 4.1 Example of duplicates

o1:Film(f12), $o1{:}director(f12,''S.Jonze'')$, $o1{:}hasActor(f12,''J.Phoenix'')$, $o1{:}hasActor(f12,''S.Johansson'')$, $o1{:}releaseDate(f12,''10/1/14'')$, $o1{:}name(f12,''Her'')$,

o1:Film(f13), $o1{:}director(f13,''S.Jonze'')$, $o1{:}director(f13,''D.Russell'')$, $o1{:}hasActor(f13,''J.Lawrence'')$, $o1{:}hasActor(f13,''B.Cooper'')$, $o1{:}releaseDate(f13,''25/12/12'')$, $o1{:}name(f13,''Her'')$

Fig. 4.2 Example containing erroneous data

4.1 Motivating example

Discovering keys in RDF datasets without taking into account possible errors or unknown duplicates may lead to lose keys. Furthermore, there may 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 Figure 4.3, we provide the OWL2 ontology $o1$ that represents the film domain. Each film can be described by its name, the release date, the language in which it was filmed, the actors and the directors involved. An expert could say that the set of properties {$o1{:}name$, $o1{:}releaseDate$} is a key for the class $Film$. When two films have the same name and the same release date, these films refer to the same film. Indeed, films sharing names are usually released in different years. Moreover, a film can be distinguished using its name and its directors. This means that even if two films have the same name, they are filmed by a different directors. Thus, the set of properties {$o1{:}name$, $o1{:}director$} could also be declared as a key by an expert.

Let us consider an approach that discovers keys automatically. A set of properties is considered as key when it uniquely identifies every distinct instance in the data. Figure 4.1 gives the RDF descriptions of two instances of the class $Film$. Note that the instances $f10$ and $f11$ refer to the same instance but no sameAs link between them has been declared.

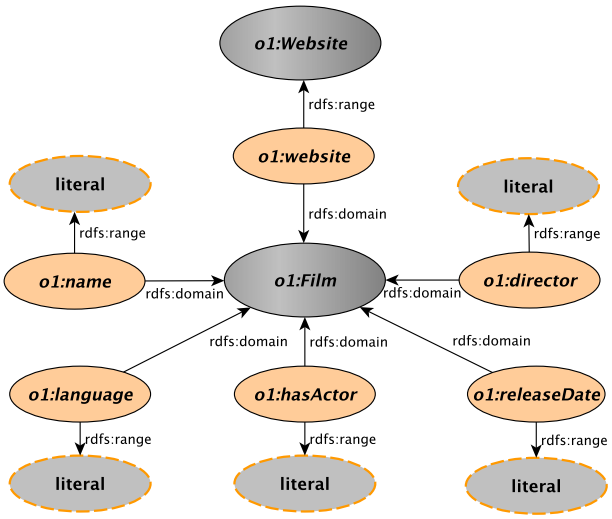


Fig. 4.3 Ontology $o1$

In this case, no key will be found. In Figure 4.2, descriptions of two films containing erroneous data are given. The film $f13$ contains two actors that do not participate in the movie while the release date provided does not correspond to the one of this film. Moreover, the director *"D.Russell"* is not directing this movie. Thus, the second description contains many erroneous values. We notice that the key $\{o1{:}name, o1{:}director\}$ cannot be discovered since both instances have the same name and share the director *"S.Jonze"*. Thus, with the presence of duplicates and errors in the data, algorithms that discover keys without exceptions might lose keys or even find no keys.

In Figure 4.4, we introduce an example containing duplicates and erroneous data, that will be used throughout this chapter, to illustrate SAKey approach. In this example, none of the two set of properties $\{o1{:}name, o1{:}releaseDate\}$ and $\{o1{:}name, o1{:}director\}$ would have been found as keys. Observing the data, we notice that there exist two films called $''Ocean's\ 12''$ and both released in the same date. In this case, we can deduce that the films $f1$ and $f6$ are either duplicates or there is an error in the name or the release date of the film.

Dataset $D1$**:**
o1:Film(f1), $o1{:}director(f1,''S.Soderbergh'')$, $o1{:}hasActor(f1,''B.Pitt'')$,
$o1{:}hasActor(f1,''J.Roberts'')$, $o1{:}releaseDate(f1,''3/4/01'')$, $o1{:}name(f1,''Ocean's\ 11'')$,
$o1{:}webSite(f1, www.oceans11.com)$

o1:Film(f2), $o1{:}director(f2,''S.Soderbergh'')$,
$o1{:}director(f2,''R.Howard'')$, $o1{:}hasActor(f2,''G.Clooney'')$, $o1{:}hasActor(f2,''B.Pitt'')$,
$o1{:}hasActor(f2,''J.Roberts'')$, $o1{:}releaseDate(f2,''2/5/04'')$, $o1{:}name(f2,''Ocean's\ 12'')$
$o1{:}webSite(f1,''www.oceans12.com'')$

o1:Film(f3), $o1{:}director(f3,''S.Soderbergh'')$,
$o1{:}director(f3,''R.Howard'')$, $o1{:}hasActor(f3,''G.Clooney'')$, $o1{:}hasActor(f3,''B.Pitt'')$
$o1{:}releaseDate(f3,''30/6/07'')$, $o1{:}name(f3,''Ocean's\ 13'')$,
$o1{:}webSite(f1, www.oceans13.com)$

o1:Film(f4), $o1{:}director(f4,''A.Payne'')$, $o1{:}hasActor(f4,''G.Clooney'')$,
$o1{:}hasActor(f4,''N.Krause'')$, $o1{:}releaseDate(f4,''15/9/11'')$,
$o1{:}name(f4,''The\ descendants'')$, $o1{:}language(f4,''english'')$
$o1{:}webSite(f1, www.descendants.com)$

o1:Film(f5), $o1{:}hasActor(f5,''D.Liman'')$,
$o1{:}releaseDate(f5,''2002'')$, $o1{:}name(f5,''The\ bourne\ Identity'')$,
$o1{:}language(f5,''english'')$
$o1{:}webSite(f1, www.bourneIdentity.com)$

o1:Film(f6),$o1{:}director(f6,''R.Howard'')$, $o1{:}releaseDate(f6,''2/5/04'')$,
$o1{:}name(f6,''Ocean's\ 12'')$

Fig. 4.4 RDF dataset $D1$

4.2 Main definitions

In this section, we provide the definitions of the main notions used in SAKey.

4.2.1 Keys with exceptions

In this chapter, we define a new notion of keys that allows exceptions, called *n-almost key*s. A set of properties is a *n*-almost key if there exist at most *n* instances that share values for this set of properties in the considered dataset.

In the dataset $D1$ of the Figure 4.4, one can notice that the property $o1$:*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 shares actors with other films as an exception, there exist four exceptions for the property $o1$:*hasActor*. We consider the property $o1$:*hasActor* as a 4-almost key since it contains at most four exceptions.

Formally, an exception represents an instance that share values with at least another instance, for a given set of properties P.

Definition 13. (Exception). *An instance X of the class c ($c \in \mathscr{C}$) is an exception for a set of properties P ($P \subseteq \mathscr{P}$) if:*

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

For example, the film $f2$ is an exception for the property $o1$:*hasActor* since "$G.Clooney$" plays also in other films.

For a class c and a set of properties P, E_P is the set of exceptions that is defined as follows:

Definition 14. (Exception set E_P). *Let c be a class ($c \in \mathscr{C}$) and P be a set of properties ($P \subseteq \mathscr{P}$). The exception set E_P is defined as:*

$$E_P = \{X \mid X \in c \text{ and } X \text{ is an exception for } P\}$$

For example, in $D1$ of Figure 4.4, we have:

$E_{\{o1:hasActor\}} = \{f1, f2, f3, f4\}$,
$E_{\{o1:hasActor,\, o1:director\}} = \{f1, f2, f3\}$.

A set of properties is considered as a n-almost key, if there exist from 1 to n exceptions in the dataset. Using the exception set E_P we give the following definition of a n-almost key.

Definition 15. (n-almost key). *Let c be a class ($c \in \mathscr{C}$), P be a set of properties ($P \subseteq \mathscr{P}$) and n an integer. P is a n-*almost key *for c if $|E_P| \leq n$.*

For example, in $D1$ the set of properties $\{o1{:}hasActor,\ o1{:}director\}$ is a 3-almost key.

By definition, a m-almost key is also a n-almost key for every $n \geq m$.

If a set of properties P is a n-almost key, every superset of P will also involve at most n exceptions, i.e., 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.

Definition 16. (Minimal n-almost key). *A set of properties P is a minimal n-*almost key *for the class c if P is a n-*almost key *and $\nexists\, P'$, a n-*almost key *$s.t.\ P' \subset P$.*

4.2.2 Discovery of n-almost keys from n-non keys

As we have already shown in the previous Chapter, an efficient way to obtain keys, is to discover first all the non keys and use them to derive the keys. Applying this idea, initially proposed in [SBHR06], SAKey derives the set of n-almost keys from the sets of properties that are not n-almost keys. Indeed, to show that a set of properties is not a n-almost key, i.e., a set of properties with at most n exceptions, it is sufficient to find at least $(n+1)$ instances that share values for this set. We call the sets that are not n-almost keys, $(n+1)$*-non keys*.

Definition 17. (n-non key). *Let c be a class ($c \in \mathscr{C}$), P be a set of properties ($P \subseteq \mathscr{P}$) and n an integer, P is a n-*non key *for c if $|E_P| \geq n$.*

4.3 SAKey Approach

The SAKey approach finds n-almost keys given an RDF dataset and a class defined in an ontology. SAKey is composed of three main steps: (1) the preprocessing steps that allow

	"J.Roberts" *"B.Pitt"* *"G.Clooney"* *"N.Krause"* *"D.Liman"*
$o1$:*hasActor*	$\{\{f1, f2\}, \{f1, f2, f3\}, \{f2, f3, f4\}, \{f4\}, \{f5\}\}$
$o1$:*director*	$\{\{f1,f2,f3\}, \{f2, f3, f6\}, \{f4\}\}$
$o1$:*releaseDate*	$\{\{f1\}, \{f2, f6\}, \{f3\}, \{f4\}, \{f5\}\}$
$o1$:*name*	$\{\{f1\}, \{f2, f6\}, \{f3\}, \{f4\}, \{f5\}\}$
$o1$:*language*	$\{\{f4, f5\}\}$
$o1$:*website*	$\{\{f1\}, \{f2\}, \{f3\}, \{f4\}, \{f5\}, \{f6\}\}$

Table 4.1 Initial map of $D1$ where values are given only for the first property

us avoid to filter the data and eliminate irrelevant sets of properties (2) the discovery of maximal (n+1)-non keys by applying pruning strategies and ordering heuristics and finally (3) an algorithm that allows the efficient derivation of n-almost keys from the set of (n+1)-non keys.

To apply this approach in every class of an ontology, SAKey can apply the strategy proposed in KD2R (see Chapter 3).

4.3.1 Preprocessing steps

Initially, we represent the descriptions of the instances of one class in a hash-map called *initial map* (see Table 4.1). Every level of the map corresponds to a property that is associated to sets of instances. Each set represents instances that share a value for this property.

We assign a unique integer to identify each instance and property of the class. The encoding of the properties is not applied in the examples of this chapter for readability reasons.

Table 4.1 shows the initial map of the dataset $D1$ presented in Figure 4.4. For example, the set $\{f2, f3, f4\}$ of the property $o1$:*hasActor* represents the films that "G.Clooney" played.

The initial map is constructed traversing the data only once.

4.3.1.1 Data filtering

To improve the scalability of our approach, we introduce two techniques to filter the data represented in the initial map, the *Singleton sets filtering.* and the *v-exception sets filtering*.

Singleton sets filtering. In the initial map, sets of size 1 represent instances that do not share

values with other instances for a given property and a given value. These sets cannot lead to the discovery of a n-non key. Thus, only non singleton sets called *v-exception sets* are kept.

Definition 18. (v-exception set E_p^v). *A set of instances $\{i_1, \ldots, i_k\}$ of the class c ($c \in \mathscr{C}$) is a E_p^v for the property $p \in \mathscr{P}$ and the value v iff $\{p(i_1, v), \ldots, p(i_k, v)\} \subseteq D$ and $|\{i_1, \ldots, i_k\}| > 1$.*

For example, in $D1$, the v-exception set of the property $o1$:*director* for the value $''Soderbergh''$ ($E^{''S.Soderbergh''}_{o1:director}$) is equal to $\{f1,\ f2,\ f3\}$.

Definition 19. (collection of v-exception sets $\mathfrak{E}_p$). *The collection of v-exception sets $\mathfrak{E}_p$ for a property $p \in \mathscr{P}$ is:*

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

For example, $\mathfrak{E}_{o1:director}$ = {{$f1$,$f2$,$f3$}, {$f2$, $f3$, $f6$}}.

Given a property p, if $\mathfrak{E}_p = \emptyset$, i.e., all the sets of p were of size 1 and have been suppressed, this property is a 1-almost key (i.e., key with no exceptions). Thus, singleton sets filtering allows the discovery of single 1-almost keys (i.e., keys composed only from one property). All these 1-almost keys are removed from the initial map and will not be considered in the n-non key discovery.

For example, in Table 4.1, we observe that there do not exist two films that share the same website. Thus, $o1$:*website* is a 1-almost key and is removed from the initial map.

v-exception sets filtering. Comparing the n-non keys that can be found thanks to two v-exception sets $E_p^{v_z}$ and $E_p^{v_m}$, where $E_p^{v_z} \subseteq E_p^{v_m}$, we can ensure that the set of n-non keys that can be found using $E_p^{v_z}$, can also be found using $E_p^{v_m}$. Indeed, $E_p^{v_z} \subseteq E_p^{v_m}$ means that $\forall i$ $p(i, v_z) \Rightarrow p(i, v_m)$. 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 $E^{''J.\ Roberts''}_{o1:hasActor}$ $\{f1, f2\}$ in the property $o1$:*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. Indeed, we discover that every time "*J. Roberts*" appears in a film also "*B, Pitt*" appears in the same film.

In this map each set of instances is sorted in an ascending order. Moreover, the sets of instances of one level are sorted as well. The set with the smallest instance identifier will be first. For example, given the sets $\{\{i4, i7\}, \{i3, i5\}, \{i3, i4, i5\}$ and $\{i3, i5\}\}$ the sorted sets will follow the order:$\{\{i3, i4, i5\}, \{i3, i5\}, \{i4, i7\}\}$.

$o1$:*hasActor*	$\{\{f1, f2, f3\}, \{f2, f3, f4\}\}$
$o1$:*director*	$\{\{f1,f2,f3\}, \{f2, f3, f6\}\}$
$o1$:*releaseDate*	$\{\{f2, f6\}\}$
$o1$:*name*	$\{\{f2, f6\}\}$
$o1$:*language*	$\{\{f4, f5\}\}$

Table 4.2 Final map of $D1$

Table 4.2 presents the data after applying the two filtering techniques on the data of the Table 4.1. This structure is called *final map* and each level represents the collection $\mathfrak{E}_p$ of the v-exception sets of a property p.

4.3.1.2 Elimination of irrelevant sets 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. Indeed, there may exist sets of properties that are never instantiated together for n common instances for different reasons. First, even if a combination of properties can theoretically be used together to describe some instances, it might never happen in a given dataset, due to the incompleteness of data. Second, there may exist distinct sets of properties that can never be used together to describe instances. 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.

The frequency of a set of properties P is the number of instances described by this set of properties. If the frequency of a set of properties is less than n, this means that, less that n instances are described by P, consequently P cannot be a n-non key. Thus, these sets of properties are irrelevant. All the relevant sets of properties can be represented by the set of maximal frequent properties. A maximal frequent set of properties is a set of properties for which none of its supersets are frequent. The problem of discovering maximal frequent sets of properties is similar than the discovery of maximal frequent itemsets in the data mining area. The complexity of this discovery is NP-hard [Yan04]. So, we aim to discover sets of properties sharing two by two, at least n instances. These sets of properties are called *potential n-non keys* (*pnk*).

Definition 20. (Potential n-non key). *A set of properties* $pnk_n = \{p_1, ..., p_m\}$ *is a* potential

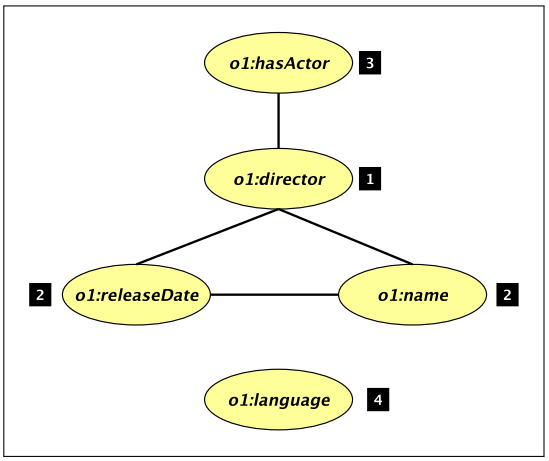


Fig. 4.5 *PNKGraph* of the dataset $D1$ when n=2

n-non key *for a class c iff:*

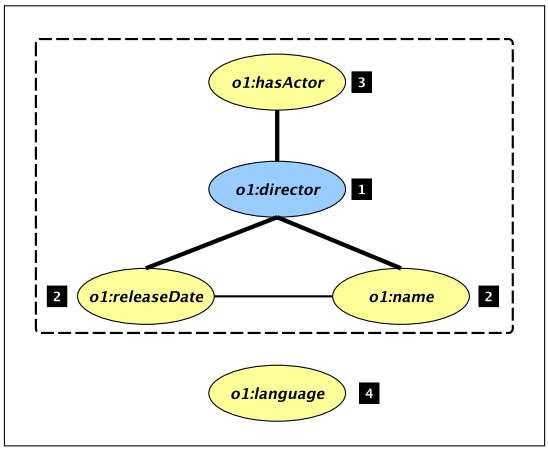
$$\forall\{p_i, p_j\} \in (pnk_n \times pnk_n) \mid I(p_i) \cap I(p_j)| \geq n$$

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

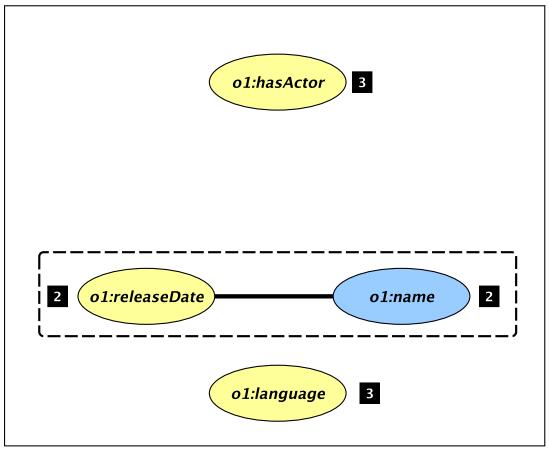
We build a graph, called *PNKGraph*, where each node represents a property and each edge between two nodes denotes the existence of at least n instances where these properties are instantiated. Figure 4.5 represents the *PNKGraph* of the data presented in the final map of the Table 4.2 when n is set to 2.

A set of properties can be a potential n-non key when for every property in this set, there is an edge to all the remaining properties. These sets are cliques i.e., sets of nodes where each pair of nodes is connected. 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*), i.e., maximal cliques. Since the problem of finding all maximal cliques of a graph is NP-Complete [Kar72], we use a greedy algorithm, inspired by the min-fill elimination order [Dec03], that discovers supersets of the maximal cliques (see Algorithm 8).

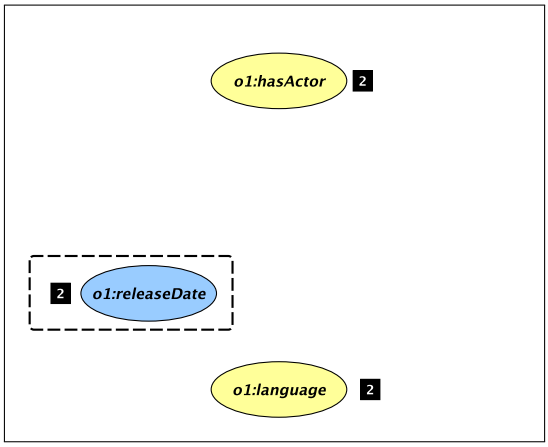
Initially, PNKFinder computes a weight for each node of a given *PNKGraph*. Given a node, the weight represents the number of edges missing between this node and all the remaining nodes in the graph. Starting from the node with the smallest weight called *nd* (see line 3), PNKFinder computes an approximate *pnk* that contains the node *nd* and all the nodes connected to this node. When the *pnk* is computed, *nd* is removed from the *PNKGraph* and the weights of its neighbors are reduced by 1. The same process is applied for all the nodes of the graph until the graph is empty.



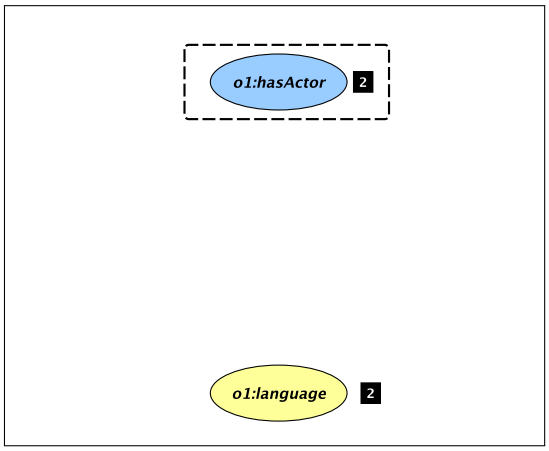
(a) Approximate *pnk* when *o*1:*director*



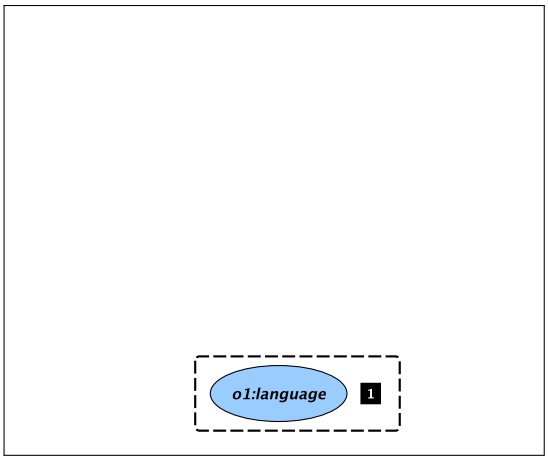
(b) Approximate *pnk* when *o*1:*name*



(c) Approximate *pnk* when *o*1:*releaseDate*



(d) Approximate *pnk* when *o*1:*hasActor*



(e) Approximate *pnk* when *o*1:*language*

Fig. 4.6 Execution of PNKFinder for the *PNKGraph* 4.5

Algorithm 8: PNKFinder

Input: *PNKGraph*
Output: *PNK*: the set of potential *n*-non keys

```
1 weightedNodes ← weightNodes(PNKGraph)
2 while PNKGraph ≠ ∅ do
3     nd ← minimumWeight(weightedNodes)
4     set ← {nd}
5     set ← set ∪ connectedNodes(nd)
6     PNK = PNK ∪ {pnk}
7     remove(nd, PNKGraph)
8     weightedNodes ← updateWeight(weightedNodes, nd)
9 PNK ← maximalSets(PNK)
```

If a clique is contained in the graph, the first time that one of this clique is chosen, all the nodes that belong to this clique will be selected. Thus, we can guarantee that each maximal clique will be included in at least one approximate *pnk*. Once all the approximate *pnk* are found, we keep only the maximal approximate *pnk*.

Example 4.3.1. Given the *PNKGraph* of the Figure 4.5, we compute the *PNK* using the algorithm PNKFinder. The weight of the node appears next to each node. Starting from the node $o1$:*director* the constructed *pnk* is {$o1$:*director*, $o1$:*hasActor*, $o1$:*releaseDate*, $o1$:*name*} (see Figure 4.6(a)). $o1$:*director* is removed from the graph and the weights are updated. Since both $o1$:*releaseDate* and $o1$:*name* have the same weight, the node containing $o1$:*name* is chosen using a lexicographical order. The *pnk* {$o1$:*name*, $o1$:*releaseDate*} is constructed (see Figure 4.6(b)). Now the node $o1$:*name* is removed and the weights are again updated. Continuing this process, the set of all *pnk* found is: {{$o1$:*director*, $o1$:*hasActor*, $o1$:*releaseDate*, $o1$:*name*}, {$o1$:*name*, $o1$:*releaseDate*}, {$o1$:*releaseDate*}, {$o1$:*hasActor*}, {$o1$:*language*}} as shown in the Figure 4.6.

4.3.2 *n*-non key discovery

This section is dedicated to the description of the *n*-non key discovery method. We first present the basic principles of the *n*-non keys discovery. Then, we introduce the pruning strategies that are used by the *nNonKeyfinder* algorithm. After that, we provide an algorithm and give an illustrative example. Finally, we introduce some property ordering and instance ordering heuristics.

4.3.2.1 Basic principles

To compute the set of maximal n-non keys, the nNonKeyFinder algorithm exploits the obtained set PNK_n of maximal potential n-non keys and the data represented in the final map.

Let us consider the property p_1 that has at least n exceptions. This property is considered as a n-non key. Intuitively, a set of properties $\{p_1,\ p_2\}$ is a n-non key iff there exist at least n distinct instances, such that each of them has the same value for p_1 and p_2 with another instance. In our framework, the sets of instances sharing the same value for a property p_1 is represented by the collection of v-exception sets $\mathfrak{E}_{p_1}$, while the sets of instances sharing the same director is represented by the collection of v-exception sets $\mathfrak{E}_{p_2}$. Intersecting each set of instances of $\mathfrak{E}_{p_1}$ with each set of instances of $\mathfrak{E}_{p_2}$ builds a new collection in which each set share values for p_1 and p_2.

More formally we introduce the *intersect operator* $\otimes$ that intersects collections of exception sets only keeping sets greater than one.

Definition 21. (Intersect operator $\otimes$). *Given two collections of v-exception sets $\mathfrak{E}_p$ and $\mathfrak{E}_{p'}$, we define the intersect $\otimes$ as follow:*

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

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 = {$o1$:*hasActor*, $o1$:*hasDirector*}, when $\mathfrak{E}_P$={{f_1, f_2, f_3}, {f_2, f_3}} the set of exceptions is E_P = {f_1, f_2, f_3} $\cup$ {f_2, f_3} = {f_1, f_2, f_3}

4.3.2.2 Pruning strategies

Computing the intersection of all the collections of v-exception sets represents the worst case scenario of finding maximal n-non keys for a set of properties. We have defined several strategies to avoid useless computations.

We will illustrate the pruning strategies in the figures Figure 4.7, Figure 4.8, Figure 4.9, and Figure 4.10. In these graphical representations each level corresponds to the collection $\mathfrak{E}_p$ of a property p and the lines express the intersections that should be computed in the worst case scenario. Thanks to the prunings, only the intersections appearing as highlighted

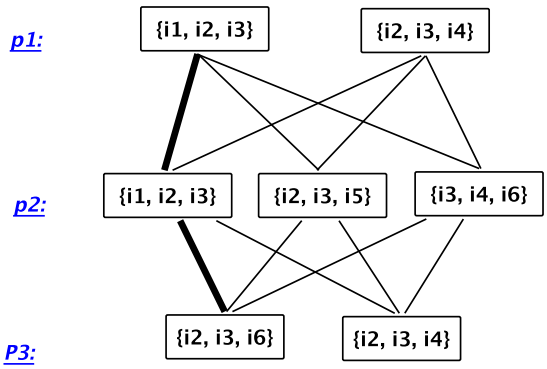


Fig. 4.7 Antimonotonic Pruning when $n = 2$

lines are computed.

Antimonotonic pruning. SAKey exploits the antimonotic characteristic of n-non keys. This pruning strategy is first introduced in KD2R in the Section 3.3.3.1. We recall that the antimonotonic pruning is based on the fact that if a set of properties is a n-non key, all its subsets are also n-non keys.

Let us consider the example of the Figure 4.7 where n is set to 2. We notice that the set of properties $\{p1,\ p2,\ p3\}$ is a 2-non key since the instances $i2$ and $i3$ share values for this set of properties. Since all the subsets of $\{p1,\ p2,\ p3\}$ are by definition also 2-non keys, no additional computation should be done.

Inclusion pruning. This strategy exploits sets inclusion to avoid computing useless intersections of v-exception sets. Given a set of properties $P = \{p_1, \ldots, p_{j-1}, p_j, \ldots, pn\}$, when the intersection of v-exception sets of $p_1, \ldots, p_{j-1}$ is included in any v-exception set of p_j only this subpath is explored (i.e., $E_{p_1}^{v_1} \cap \ldots \cap E_{p_{j-1}}^{v_{j-1}} \subseteq E_{p_j}^{v_i}$). This means that $p_1(i, v_i) \wedge \ldots \wedge p_{j-1}(i, v_{j-1}) \Rightarrow p_j(i, v_j)$ for every instance i.

For example, in Figure 4.8, we notice that the only 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 v-exception sets 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. In this example, we discover that $\forall i\ p_1(i, v_1) \Rightarrow p_2(i, v_2)$.

Seen intersection pruning. When a new intersection is included in an already computed intersection, the exploration using the new intersection cannot lead to new n-non keys. Thus,

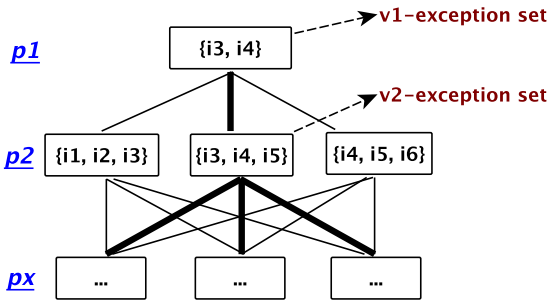


Fig. 4.8 Inclusion Pruning

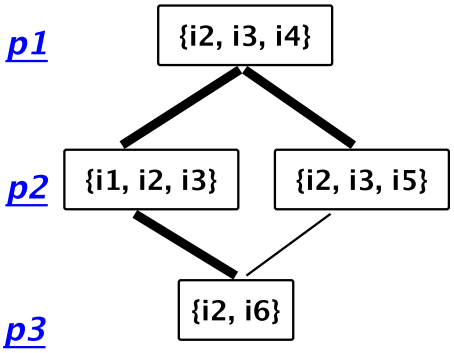
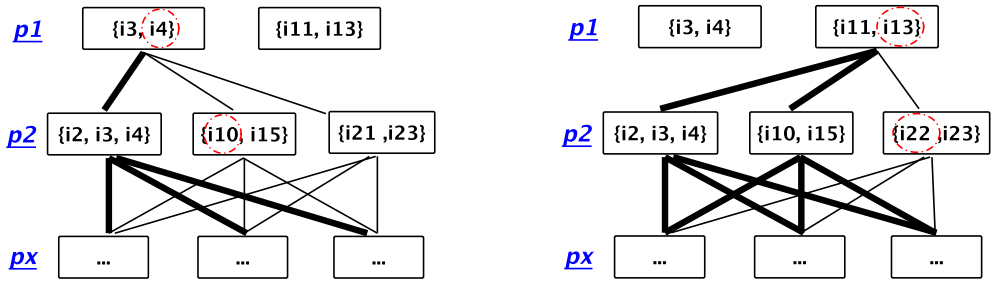


Fig. 4.9 Seen Intersection Pruning

this exploration should not continue.

In Figure 4.9, we observe that starting from the v-exception set of the property p_1 $\{i_2, i_3, i_4\}$, the intersection between this v-exception set and $\{i_1, i_2, i_3\}$ or $\{i_2, i_3, i_5\}$ of the property p_2, 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.

Irrelevant intersection pruning. Assigning a unique integer to each instance of the final map and sorting the v-exception sets of each collection allows us to avoid useless intersections. The instances of a v-exception set are sorted in ascending order, from the smallest to the biggest. The v-exception sets of a collection are sorted as well. We sort two v-exception sets by comparing instances in the same position. This comparison is done instance by instance until one v-exception set to be found as smaller than another. The first position where two v-exception sets have different integers will determine their order. The v-exception set with the smallest integer will be first. For example, given the v-exception sets $\{\{i4, i7\},$



(a) (b) Irrelevant intersections

Fig. 4.10 *n*NonKeyFinder prunings and execution

$\{i3, i5\}$, $\{i3, i4, i5\}$ and $\{i3, i5\}\}$ the sorted v-exception sets will follow the order: $\{\{i3, i4, i5\}, \{i3, i5\}, \{i4, i7\}\}$.

The intersection between a current intersection k and each v-exception set of a collection $\mathfrak{E}_p$ of a property p can stop when the last element of k is smaller or equal to the first element of a v-exception set m of $\mathfrak{E}_p$. Indeed, when the last element of k is smaller than the first element of m, all the v-exception sets of the property p appearing after the v-exception set m will contain instances with bigger identifiers. Thus, their intersection with k will be by definition empty.

In Figure 4.10, we notice that the intersections of the v-exception set $\{i3, i4\}$ of the property p_1 with the v-exception sets of the property p_2 will stop when the v-exception set $\{i10, i15\}$ is found since we can guarantee that all the v-exception sets found after the v-exception set $\{i10, i15\}$ will contain instances that are not included in the selected v-exception set.

4.3.2.3 *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 $E_P \geq n$ this exploration stops as soon as n exceptions are found.

The *n*NonKeyFinder algorithm (see Algorithm9) 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 *n*NonKeyFinder is: *n*NonKeyFinder(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 of one class.

Given the union of a property p_i and the *curNKey*, which corresponds to the set of already explored properties, the function *uncheckedNonKeys* ensures that this set of properties should be explored. More precisely, this function 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 current intersection, $curInter$, is included in one of the v-exception sets of the property p_i, the $selectedE_p^v$ will contain only the $curInter$. Otherwise, all the v-exception sets of 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 18). 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 21), forgetting the property p_i in order to explore all the possible combinations of properties.

The data of the final map presented in Table 4.2 appear in the Figure 4.11. Table 4.3 shows the execution of nNonKeyFinder for the example of the Figure 4.11 where $PNK = \{\{o1{:}hasActor,\ o1{:}director,\ o1{:}releaseDate\},\ \{o1{:}language\}\}$. For clarity reason, we rename the properties as it follows : $p_1 = o1{:}hasActor$, $p_2 = o1{:}director$, $p_3 = o1{:}releaseDate$, $p_4 = o1{:}name$, $p_5 = o1{:}language$.

4.3.2.4 Ordering heuristics

We present two additional ordering heuristics that can be applied to improve the scalability of th n-non key discovery. The first heuristic concern the order with which the properties are explored. Furthermore, we present the ordering of instances in the final map.

Properties ordering. The order in which the properties are explored by the nNonKeyFinder algorithm, affects the number of computations needed to discover all the maximal n-non keys. The goal is to eliminate as fast as possible, combinations of properties that cannot

p_i	$selectedE_p^v$	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	$\{5\}$	5	$\{f1,f2,f3\}$	$\{p_1,p_2\}$	$\{\}$	$\{\{p_1\},\{p_1,p_2\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)\}$
p_4	$\{6\}$	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_5	-	-	$\{f1,f2,f3\}$	$\{p_1,p_2\}$	$\{\}$	$\{\{p_1\},\{p_1,p_2\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)\}$
p_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_2	$\{3,4\}$	3	$\{f_2,f_3,f_4\}$	$\{p_1\}$	$\{\{f_1,f_2,f_3\}\}$	$\{\{p_1\},\{p_1,p_2\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)\}$
p_3	$\{5\}$	5	$\{f_2,f_3,f_4\}$	$\{p_1\}$	$\{\{f_1,f_2,f_3\}\}$	$\{\{p_1\},\{p_1,p_2\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)\}$
p_4	$\{6\}$	6	$\{f_2,f_3,f_4\}$	$\{p_1\}$	$\{\{f_1,f_2,f_3\}\}$	$\{\{p_1\},\{p_1,p_2\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)\}$
p_5	-	-	$\{f_2,f_3,f_4\}$	$\{p_1\}$	$\{\{f_1,f_2,f_3\}\}$	$\{\{p_1\},\{p_1,p_2\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)\}$
p_2	$\{3,4\}$	3	$\{f1,\ldots,f6\}$	$\{\}$	$\{\{f_1,f_2,f_3\},\{f_2,f_3,f_4\}\}$	$\{\{p_1\},\{p_1,p_2\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)$ $(p_2):(f_1,f_2,f_3)\}$
p_2	$\{3,4\}$	4	$\{f1,\ldots,f6\}$	$\{\}$	$\{\{f_1,f_2,f_3\},\{f_2,f_3,f_4\}\}$	$\{\{p_1\},\{p_1,p_2\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)$ $(p_2):(f_1,f_2,f_3,f_6)\}$
p_3	$\{5\}$	5	$\{f2,f3,f6\}$	$\{p_2\}$	$\{\{f_1,f_2,f_3\},\{f_2,f_3,f_4\}\}$	$\{\{p_1\},\{p_1,p_2\},$ $\{p_2,p_3\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)$ $(p_2):(f_1,f_2,f_3,f_6)$ $(p_2,p_3):(f_2,f_6)\}$
p_4	$\{6\}$	6	$\{f2,f6\}$	$\{p_2\}$	$\{\{f_1,f_2,f_3\},\{f_2,f_3,f_4\}\}$	$\{\{p_1\},\{p_1,p_2\},$ $\{p_2,p_3\}$ $,\{p_2,p_3,p_4\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)$ $(p_2):(f_1,f_2,f_3,f_6)$ $(p_2,p_3):(f_2,f_6)$ $(p_2,p_3,p_4):(f_2,f_6)\}$
p_5	-	-	$\{f2,f6\}$	$\{p_2\}$	$\{\{f_1,f_2,f_3\},\{f_2,f_3,f_4\}\}$	$\{\{p_1\},\{p_1,p_2\},$ $\{p_2,p_3\}$ $,\{p_2,p_3,p_4\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)$ $(p_2):(f_1,f_2,f_3,f_6)$ $(p_2,p_3):(f_2,f_6)$ $(p_2,p_3,p_4):(f_2,f_6)\}$
p_3	-	-	$\{f2,f6\}$	$\{p_2\}$	$\{\{f_1,f_2,f_3\},\{f_2,f_3,f_4\},$ $\{f_2,f_3,f_6\}\}$	$\{\{p_1\},\{p_1,p_2\},$ $\{p_2,p_3\}$ $,\{p_2,p_3,p_4\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)$ $(p_2):(f_1,f_2,f_3,f_6)$ $(p_2,p_3):(f_2,f_6)$ $(p_2,p_3,p_4):(f_2,f_6)\}$
p_4	-	-	$\{f2,f6\}$	$\{p_2\}$	$\{\{f_1,f_2,f_3\},\{f_2,f_3,f_4\}\}$ $\{f_2,f_3,f_6\}\}$	$\{\{p_1\},\{p_1,p_2\},$ $\{p_2,p_3\}$ $,\{p_2,p_3,p_4\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)$ $(p_2):(f_1,f_2,f_3,f_6)$ $(p_2,p_3):(f_2,f_6)$ $(p_2,p_3,p_4):(f_2,f_6)\}$
p_5	$\{7\}$	7	$\{f1,\ldots,f6\}$	$\{\}$	$\{\{f_1,f_2,f_3\},\{f_2,f_3,f_4\}\}$ $\{f_2,f_3,f_6\}\}$	$\{\{p_1\},\{p_1,p_2\},$ $\{p_2,p_3\}$ $\{p_2,p_3,p_4\},\{p_5\}\}$	$\{(p_1):(f_1,f_2,f_3)$ $(p_1,p_2):(f_1,f_2,f_3)$ $(p_2):(f_1,f_2,f_3,f_6)$ $(p_2,p_3):(f_2,f_6)$ $(p_2,p_3,p_4):(f_2,f_6)$ $(p_5):(f_4,f_5)\}$

Table 4.3 *n*NonKeyFinder execution on the example of Figure 4.11

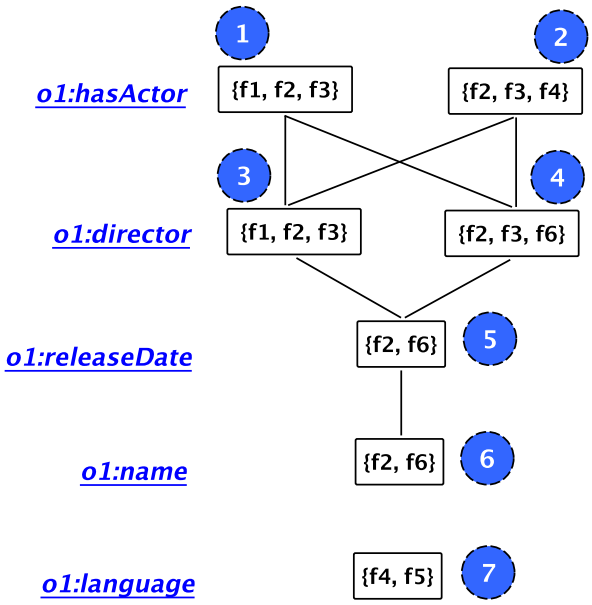


Fig. 4.11 example of *n*NonKeyFinder

lead to *n*-non keys. The intuition is that properties containing small v-exception sets (i.e., v-exception sets composed of few instances) will be less intersected with other properties since they can arrive faster to intersections with size 0 or 1. Since this is the stopping condition of the algorithm, starting from the properties containing small v-exception sets we manage to eliminate many useless intersections. Indeed, ordering the properties prevent us from intersecting properties that do not shared by many instances. Properties having values that are shared by many instances are left for the end.

Each property is assigned with a value that corresponds to the size of its biggest v-exception set in terms of number of instances. For example, if a property p contains the following v-exception sets, $\mathfrak{E}_p = \{\{i1, i2, i4\}, \{i3, i5, i6, i7, i8\}, \{i10, i15\}\}$, the property p is assigned with the value 5 which represents the size of the v-exception set $\{i3, i5, i6, i7, i8\}$.

Thus, the properties are sorted in ascending order (i.e., from the smallest to the biggest).

Instances ordering. Exploring instances that appear more in the v-exception sets first, ensures that *n*-non keys might be discovered more efficiently. We call these instances frequent. To find *n*-non keys fast, we assign small identifiers to frequent instances. This allows us to check first the frequent instances.

Algorithm 9: nNonKeyFinder

Input : p_i: a current property
$curInter$: a current intersection
$curNKey$: already explored set of properties
$seenInter$: computed intersections
$nonKeySet$: set of discovered n-non keys
E: the set of exceptions
n: number of exceptions

Output: $nonKeySet$: set of discovered n-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^v_{p_i} ← {{curInter}}
5      else
6          selectedE^v_{p_i} ← 𝔈_{p_i}
7      foreach E^v_{p_i} ∈ selectedE^v_{p_i} do
8          if (last(curInter) ≤ first(E^v_{p_i})) //Irrelevant Intersection Pruning then
9              break
10         newInter ← E^v_{p_i} ∩ curInter
11         if (|newInter| > 1) then
12             if (newInter ⊈ k s.t. k ∈ seenInter) //Seen Intersection Pruning then
13                 nvNKey ← {p_i} ∪ curNKey
14                 update(E, nvNKey, newInter)
15                 if (|E_{nvNkey}| > n) then
16                     nonKeySet ← nonKeySet ∪ {nvNKey}
17                 if ((i+1) < # properties) then
18                     nNonKeyFinder(p_{i+1}, newInter, nvNKey, seenInter, nonKeySet, E, n)
19         seenInter ← seenInter ∪ {newInter}
20 if ((i+1) < # properties) then
21     nNonKeyFinder(p_{i+1}, curInter, curNKey, seenInter, nonKeySet, E, n)
```

Instances are sorted by their occurrence globally in the final map. The more an instance appears in v-exception sets, the smaller value it will be assigned with.

4.3.3 Ordered key derivation

Following the idea of key derivation presented in Section 3.3.3.3, the set of minimal n-almost keys can be obtained using the sets of all the sets of properties that are not n-almost

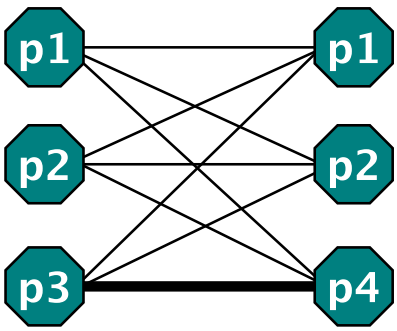


Fig. 4.12 Minimum computations to obtain the minimal keys

keys. A set of properties is a n-almost key if it contains at most n exceptions. Thus, a set of properties is considered as not a n-almost key when it contains more than n exceptions, i.e., (n+1)-non key.

As we have seen in the previous chapter (see Section 3.3.3.3), both KD2R and [SBHR06] derive the set of keys using the set of maximal non 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 same algorithm can be applied in our case. Nevertheless, the complexity of computing the Cartesian product 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.

For example, let us consider the set of properties $\{p1,\ p2,\ p3,\ p4\}$ for a given class and its maximal (n+1)-non keys $\{\{p3\},\ \{p4\}\}$. The complement sets of the two (n+1)-non keys are $\{p1,\ p2,\ p3\}$, $\{p1,\ p2,\ p4\}$ respectively. The Cartesian product of the complement sets is:

$\{p1,\ p2,\ p3\} \times \{p1,\ p2,\ p4\} = \{\{p1,\ p1\},\ \{p1,\ p2\},\ \{p1,\ p3\},\ \{p2,\ p1\},\ \{p2,\ p2\},$
$\{p2,\ p4\},\ \{p3,\ p1\},\ \{p1,\ p2\},\ \{p3,\ p4\}\}.$

Once the Cartesian product is computed, the simplification step is applied and the following minimal n-almost keys are obtained:

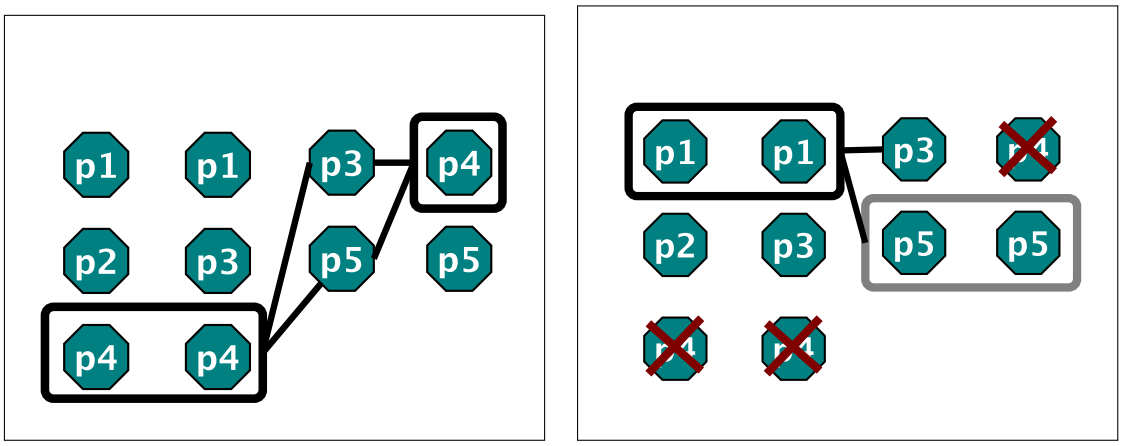
n-almost keys = $\{\{p1\},\ \{p2\},\ \{p3,\ p4\}\}$.

First, we can observe that keys do not contain duplicate properties. In the previous example, the set $\{p1,\ p1\}$ should not be constructed. Moreover, the order of properties is not relevant.

Thus, if the set $\{p1,\ p2\}$ is constructed, $\{p2,\ p1\}$ is not necessary. Finally, let us consider a set of properties $\{p1,\ldots,pi\}$ that corresponds to a part of a n-almost key constructed from properties found belonging in different complement sets. If pi and pj (or any other property in the current set) belong to a not already considered complement set, if pj is added to $\{p1,\ldots,pi\}$, we are sure that this new set of properties will never lead to a minimal n-almost key. For example, considering the property $p1$ of the complement set $\{p1,\ p2,\ p3\}$, the set $\{p1,\ p2\}$ should be not be constructed since $p1$ is contained also in $\{p1,\ p2,\ p4\}$.In Figure 4.12, the only highlighted edge corresponds to the only necessary combination of properties in order to compute the n-almost key $\{p3,\ p4\}$.

To avoid useless computations, we propose a new algorithm that derives fast minimal keys, called OrderedKeyDerivation (see Algorithm 10) from the set of complement sets. The properties are sorted according to their frequency in the complement sets. The frequency of one property corresponds to the number of complement sets containing this property. At each iteration, the most frequent property is selected and all the keys involving this property are discovered recursively. To avoid constructing non-minimal keys, we combine the most frequent property p only with properties found in the complement sets not containing p. Once all the keys including this property are found, the selected property is eliminated from every complement set. In this way, the size of complement sets decreases in each iteration, thus less computations will be made. The process continues for every property until one complement set becomes empty. Since the condition for producing a valid key is to contain at least one property from each complement set, when the deletion of a property leads to an empty complement set, all the n-almost keys have been discovered and no new keys can be produced. When all the n-almost keys are found a minimization step has to be applied to ensure that only minimal keys will be kept.

Example 4.3.2. Let the set of $\mathscr{P}$ be $\{p_1, p_2, p_3, p_4, p_5\}$. 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 will be $\{\{p_1, p_5\},\{p_2, p_3, p_5\},\{p_3, p_4\},\{p_4, p_5\}\}$. The process is shown in the Figure 4.13 and is explained in details in the following.
Initially the algorithm takes as input the following complement sets: $\{\{p_1, p_2, p_4\}$, $\{p_1, p_3, p_4\}$, $\{p_3, p_5\}$, $\{p_4, p_5\}\}$. At this point the *keySet* is empty. The properties are explored in the order $\{p_4, p_1, p_3, p_5, p_2\}$. Starting from the most frequent property, p_4, we calculate all the keys containing this property. *selectedCompSets* corresponds to the complement sets not containing this property. In this case, $\{p_3, p_5\}$ is the only complement set that that does not contain the property p_4. The property p_4 is combined with every property of this set. To do so, the OrderedKeyDerivation is now called with $\{\{p_3, p_5\}\}$.



(a) Step 1 - when property is p_4

(b) Step 2 - when property is p_1

(c) Step 3 - when property is p_5

(d) Step 4 - Stopping condition

Fig. 4.13 OrderedKeyDerivation of the Example 4.3.2

The properties p_3 and p_5 are now lexicographically ordered since they both have frequency 1. Starting from the property p_3, there is no complement set that does not contain this property. Thus, *selectedCompSets* is empty. In this case, the set $\{p_3\}$ is added to the *keySet*. The property p_3 is removed from the *selectedCompSets* which is now $\{\{p_5\}\}$. Continuing with the property p_5, the *selectedCompSets* is again empty. The *keySet* is now $\{\{p_3\}, \{p_5\}\}$. By removing p_5 from the *compSets*, *compSets* becomes $\{\{\}\}$. At this point, the Cartesian product of p_4 with $p3$ and p_5 is computed. Now the *keySet* is $\{\{p_3, p_4\}, \{p_4, p_5\}\}$. After that, the property p_4 is eliminated and the *compSets* are: $\{\{p_1, p_2\}, \{p_1, p_3\}, \{p_5\}, \{p_3, p_5\}\}$. The most frequent properties now are p_1, p_3 and p_5, having all a frequency of 2. In this case, we order lexicographically the properties, thus we select the property p_1. OrderedKeyDerivation is now called with the *selectedCompSets* which are are now $\{\{p_3, p_5\}, \{p_5\}\}$. In this case, the property p_5 is more frequent, thus we start with this. There is no complement set not containing p_5, so the *keySet* $= \{p_5\}$.

Removing this property, the *compSets* is now $\{\{p_3\},\{\}\}$. We observe that one of the sets is empty, so the for loop stops. In this case, the Cartesian product of $\{p_1\}$ with $\{p_5\}$ is computed and added to the *keySet*. The *keySet* is now $\{\{p_3, p_4\},\{p_4, p_5\}, \{p_1, p_5\}\}$. The *compSets* after removing the property p_1 are: $\{\{p_2\},\{p_3\},\{p_5\},\{p_3, p_5\}\}$. We continue with the property p_3. The *selectedCompSets* is $\{\{p_2\},\{p_5\}\}$. The OrderedKeyDerivation is now called with p_2 and since the *selectedCompSets* is $\{\{p_5\}\}$ the OrderedKeyDerivation is recursively called for p_5. In this case, the *keySet* is initially $\{\{p_5\}\}$, it becomes $\{\{p_2,\ p_5\}\}$ and finally $\{\{p_2,\ p_3,\ p_5\}\}$. When the property p_3 is removed from the *compSets*, one of the sets becomes empty. Thus, at this point all the n-almost keys have been discovered. Finally, the discovered *keySet* is $\{\{p_1, p_5\},\{p_2, p_3, p_5\},\{p_3, p_4\},\{p_4, p_5\}\}$. In this example, we we observe that all the discovered n-almost keys correspond to minimal ones, thus the minimization step will not change the result found so far.

To conclude, if every property has a different frequency in the complement sets, this algorithm discovers directly only minimal n-almost keys. In the case where there exist properties having the same frequency, non minimal keys might be derived. When all the keys are found, an extra step is required in order to simplify the set of keys and have in the end only non minimal n-almost keys. The worst case complexity of OrderedKeyDerivation is $O(k^m)$ only when every property in the complement sets is contained in only one complement set.

Algorithm 10: OrderedKeyDerivation

Input: *compSets*: set of complement sets
Output: *keySet*: set of n-almost keys

```
1  keySet ← ∅
2  orderedProperties = getOrderedProperties(compSets)
3  foreach p_i ∈ orderedProperties do
4      selectedCompSets ← selectSets(p_i, compSets) //compSets not containing p_i
5      if selectedCompSets == ∅ then
6          keySet = keySet ∪ {{p_i}}
7      else
8          foreach curKey ∈ OrderedKeyDerivation(selectedCompSets) do
9              keySet = KeySet ∪ {{p_i} × curKey}
10     compSets = remove(compSets, p_i)
11     if ∃ set ∈ compSet | set == ∅ then
12         break
13 return keySet
```

4.4 Valid almost keys in different datasets

Let us consider two datasets conforming to distinct ontologies. In the case of n-almost keys is also interesting to find valid n-almost keys for several datasets. We present the merge of n-almost keys in two different scenarios. In the first scenario the number of exceptions is equal in both datasets. In this case n-almost keys will be found in both datasets and will be merged in the same way as presented in Section 3.3.3.4. This means that the merged almost keys will correspond to the n-almost keys valid in both datasets.

In the second scenario where almost keys are found for different n in each dataset, the merged results as presented in Section 3.3.3.4 will correspond to the biggest number of exceptions used in this datasets.

4.5 C-SAKey: Conditional key discovery

In order to enrich, as much as possible, the set of keys that can be declared for a specific domain, we propose C-SAKey, an extension of SAKey that discovers conditional keys. A set of properties is a conditional key for a class, when it is a key for the instances of the class that satisfy a given condition. Here, we consider conditions that involve one or several datatype properties for which a value is specified. More precisely, given a class c ($c \in \mathscr{C}$), an instance X and the set of properties $P = \{p_1, \ldots, p_m\}$ where $P \in \mathscr{P}$, a condition $Cond(X)$ can be expressed as:

$$p_1(X, v_1) \wedge \ldots \wedge p_m(X, v_m)$$

For example, let us consider the RDF dataset $D1$ presented in the Figure 4.14. This dataset contains descriptions of researchers. We observe that there exist two researchers that have the last name $''Sais''$ therefore the property *lastName* is not a key for the class *Researcher*. In a bigger dataset many researchers can share last names. Moreover, $\{lastName, worksIn\}$ is not a key since there can exist people that work in the same lab and share last names. However, there may exist research labs where every researcher can be identified by its last name. Therefore, it is interesting to discover keys that are valid in subparts of the data that are selected using a specific condition. In this example, we observe that the property *lastName* is a key for the researchers that work in $''LRI''$, while it is not for the researchers in $''CRIL''$.

In the previous chapters, we focus on the discovery of keys for a given class. OWL2 allows the declaration of conditional keys using a *class expression CE*. As already seen in

Dataset *D1*:
Researcher(r1), $firstName(r1, "Fatiha")$, $lastName(r1, "Sais")$, $worksIn(r1, "LRI")$, $position(r1, "Assist.professor")$

Researcher(r2), $firstName(r2, "Nathalie")$, $lastName(r2, "Pernelle")$, $worksIn(r2, "LRI")$, $position(r1, "Assist.professor")$,

Researcher(r3), $firstName(r3, "Chantal")$, $lastName(r3, "Reynaud")$, $worksIn(r3, "LRI")$, $position(r3, "Professor")$,

Researcher(r4), $firstName(r4, "Lakhdar")$, $lastName(r4, "Sais")$, $worksIn(r4, "CRIL")$, $position(r4, "Professor")$

Researcher(r5), $firstName(r5, "Olivier")$, $lastName(r5, "Roussel")$, $worksIn(r5, "CRIL")$, $position(r5, "Assist.professor")$

Researcher(r6), $firstName(r6, "Michel")$, $lastName(r6, "Roussel")$, $worksIn(r6, "CRIL")$, $position(r6, "Engineer")$

Fig. 4.14 RDF dataset $D1$

Section 2.1.2, a OWL2 key can be represented as $(CE(ope_1,\ldots,ope_m)\ (dpe_1,\ldots,dpe_n))$ where ope represent the object properties and dpe the datatype properties. A class expression represents instances of a class that fulfill a set of specific constraints that can be declared using various OWL2 constructs [1]. To build the conditions that we consider, the construct *owl:DataHasValue* can be used.

We consider that a conditional key is valid in a considered dataset if the following definition is satisfied:

Definition 22. (Conditional key). *A set of properties P ($P \subseteq \mathscr{P}$) is a conditional key for the class c and the condition Cond in a dataset D if:*

$$\forall X\ \forall Y\ ((X \neq Y) \wedge c(X) \wedge c(Y) \wedge Cond(X) \wedge Cond(Y)) \Rightarrow$$

$$(\forall Z\ \neg(p_j(X,Z) \wedge p_j(X,Z)))$$

Discovering conditional keys for all the possible conditions that can be expressed for a given dataset would be very time consuming. Moreover, conditional keys that are true

[1] See http://www.w3.org/TR/owl2-syntax/#Class_Expressions for more details

only in small subparts of the data may not be significant. We consider that an expert selects sets of properties that are going to be used in the condition. For each set of properties, all the combinations of values are used to generate the set of conditions to be explored. A preprocessing step is applied to extract all the descriptions of instances that contain values involved in the condition. A naive way to discover conditional keys, is then, to apply the Algorithm 9, used in SAKey to discover n-non keys, to these extracted descriptions. Note that, the value of n is set to 1 since we are interested in the discovery of exact conditional keys, i.e., conditional keys with no exceptions.

4.6 Experiments

We evaluated SAKey using three 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 to 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 the use of n-almost keys can improve the quality of data linking. The experiments are executed on three different datasets, DBpedia[2], YAGO[3] (knowledge base presented in [SKW07]), OAEI 2010[4] and OAEI 2013[5]. The execution time of each experiment corresponds to the average execution time of 10 executions.

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 and processor 2x2.4Ghz, 6-Core Intel Xeon running Mac OS X 10.8.

4.6.1 Scalability of SAKey when $n = 1$

SAKey has been executed on the set of instances of every top class of DBpedia ($\simeq$400 classes). Here we provide details about the scalability of SAKey only on the classes $DB{:}NaturalPlace$, $DB{:}BodyOfWater$ and $DB{:}Lake$ of DBpedia (see Figure 4.7) when n = 1. We first compare the size of the considered data before and after the filtering steps. Then we give the results of SAKey when it is run on filtered data with and without the prunings.

[2]http://wiki.dbpedia.org/Downloads39
[3]http://www.mpi-inf.mpg.de/yago-naga/yago/downloads.html
[4]http://oaei.ontologymatching.org/2010/im/index.html
[5]http://oaei.ontologymatching.org/2013

class	**# Initial sets**	**# Final sets**	
DB:*Lake*	57964	4856 (8.3%)	
DB:*BodyOfWater*	139944	14833 (10.5%)	
DB:*NaturalPlace*	206323	22584 (11%)	
class	**# filtered singleton sets**	**# filtered E_p^v**	**# filtered 1-Almost keys**
DB:*Lake*	50807	2301	78 (54%)
DB:*BodyOfWater*	120949	4162	120 (60%)
DB:*NaturalPlace*	177278	6461	131 (60%)

Table 4.4 Data filtering in different classes of DBpedia

In the rest of this section, we provide the execution time for the biggest class of DBpedia, *DB*:*Person*.

Data filtering experiment. To validate the importance of our filtering steps, we compare the number of sets of instances before and after the filtering steps. In Table 4.4, the initial sets correspond to the sets of instances represented in the initial map while the final sets correspond to the sets remained after (i) the singleton sets filtering and (ii) the v-exception sets filtering, presented in Section 4.3.1.1. The singleton sets refer to the sets of size 1 removed from the final map, thanks to the singleton sets filtering. 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. The suppressed properties represent the properties that contain only sets of size 1, thus they cannot be considered as n-non keys. SAKey is able to discover directly single 1-almost keys, i.e., keys with no exceptions composed of one property. More than 50% of the properties of each class are suppressed since they are single 1-almost keys ($\mathfrak{E}_p = \emptyset$ after the singleton sets filtering). For example, the property *mouthPosition* is a 1-almost key for the class *DB*:*Lake*.

Pruning Experiment. To validate the importance of the pruning strategies, we run nNonKeyFinder on different datasets with and without applying the pruning strategies. In Table 4.5, we compare the number of calls of nNonKeyFinder along with the time, when the algorithm exploits the following pruning strategies (see Section 4.3.2.2): the anti-monotonic pruning, the inclusion pruning, the seen intersection pruning and the irrelevant intersection pruning. As we observe, the number of calls of nNonKeyFinder decreases significantly using the prunings. The percentages represent ratio of the number of calls when prun-

ings are used, to the number of calls when no pruning is used. In the class *DB*:*Lake*, the number of calls of SAKey decreases to half when prunings are used, while in the class *DB*:*NaturalPlace* only 20% of the calls without prunings are done when prunings are used. Consequently, the runtime of SAKey improves significantly. For example, in the class *DB*:*NaturalPlace* the time decreases by 76%.

We observe that thanks to the use of filtering techniques and the pruning strategies, SAKey manages to scale, avoiding many useless computations.

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 4.5 Prunings and execution time of *n*nonKeyFinder on classes of DBpedia

4.6.2 Scalability of SAKey when $n > 1$

To evaluate the scalability of the n-non key discovery, we run SAKey for different values of n. In Table. 4.6, we notice that the execution time of *n*NonKeyFinder for the class *DB*:*NaturalPlace* is not strongly affected by the increase of n. Additionally, we notice that the number of *n*NonKeyFinder calls increases very slowly. For example, comparing the number of calls of *n*NonKeyFinder when $n = 1$ and $n = 300$, only 57 extra calls are necessary to obtain all the 300-non keys, i.e., non keys containing at least 300 exceptions. Of course, setting the n up to 300 will not lead to the discovery of significant keys. The values of n have been set so high only to prove the ability of SAKey to scale, even when the number of allowed exceptions is big.

4.6.3 KD2R vs. SAKey

In this section, we compare SAKey to KD2R in two steps. The first experiment compares the efficiency of SAKey against KD2R in the non key discovery process. Since KD2R cannot discover keys with exceptions, the value of n is set to 1. Given the same set of non keys, the second experiment compares the runtime of the key derivation approach of KD2R to our novel algorithm.

n	**# of *n*NonKeyFinder Calls**	***n*NonKeyFinder Runtime**	**# *n*-non keys**
1	257150	61s	298
50	257207	55s	118
100	257207	58s	78
200	257207	56s	53
300	257207	59s	45
400	257260	56s	41

Table 4.6 *n*NonKeyFinder applying different *n* values on *DB*:*NaturalPlace*

Non key discovery results. In Figure 4.15, we compare the runtimes of the non key discovery of both KD2R and SAKey for the class *DB*:*NaturalPlace*. In this experiment, we want to compare the resistance of both algorithms to the number of properties. 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. Additionally, KD2R cannot discover the non keys when the complete set of properties is selected since the algorithm demands a big memory space. Note that when only 10 properties are selected, KD2R is faster than SAKey by few seconds. Indeed, this happens since SAKey applies many preprocessing steps to avoid useless computations.

In Figure 4.15, we notice that SAKey is linear in the beginning and almost constant after a certain number of properties. This happens since the class *DB*:*NaturalPlace* contains many properties which appear only in few instances. Since the properties are added according to their frequency, the properties added in the last tests contain few exception sets. It occurs also that many of the properties added in the end are single keys and unlike KD2R, SAKey is able to discover them directly using the singleton sets pruning.

To show that SAKey outperforms KD2R, we run both algorithms in several classes of DBpedia and YAGO. Table. 4.7 provides information for each class such as the number of triples, the number of instances and the number of properties. In this table, we observe that SAKey is orders of magnitude faster than KD2R in every exploited class. For example, SAKey needs only 11 seconds to discover the set of non keys for the class *DB*:*Mountain*, while KD2R needs more than 119 minutes. Moreover, KD2R runs out of memory in classes containing many properties and triples.

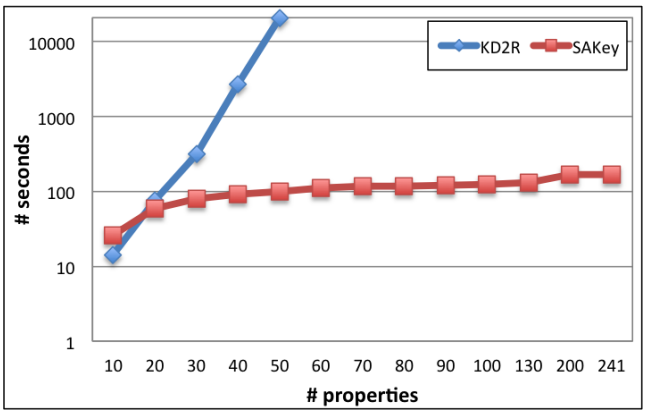


Fig. 4.15 KD2R vs. SAKey: Non key discovery runtime on *DB*:*NaturalPlace*

class	# triples	# instances	# properties	KD2R Runtime	SAKey Runtime
DB:*Website*	8506	2870	66	13min	1s
YA:*SportsSeason*	83944	17839	35	2min	9s
YA:*Building*	114783	54384	17	26s	9s
DB:*Mountain*	115796	12912	124	119min	11s
DB:*Lake*	409016	9438	111	outOfMem.	8s
DB:*BodyOfWater*	1068428	34000	200	outOfMem.	37s
DB:*NaturalPlace*	1604348	49913	243	outOfMem.	1min10s

Table 4.7 Runtime of *n*NonKeyFinder in different classes of DBpedia and YAGO

In the biggest class of DBpedia, *DB*:*Person* (more than 8.000.000 triples, 900.000 instances and 508 properties), SAKey takes 19 hours to compute the *n*-non keys while KD2R cannot even be applied. Indeed, unlike KD2R, SAKey uses a very compressed representation of the data and applies new pruning strategies that allow it to scale even on big datasets.

Key derivation results. Given different sets of non keys, we compare the runtimes of the key derivation algorithms of KD2R and SAKey for the class *DB*:*BodyOfWater*. Given the complete set of non keys of the class *DB*:*BodyOfWater*, we run both algorithms in different subsets of the non keys set. Starting from 20 non keys, we randomly add non keys and extract keys with both algorithms. The Figure 4.16 shows how the time evolves when the number of non keys of the class *DB*:*BodyOfWater* increases. We observe that SAKey outperforms KD2R in every case. For example, when the number of non keys is 190, KD2R needs more than 1 day to compute the set of minimal keys, while SAKey takes less than 1

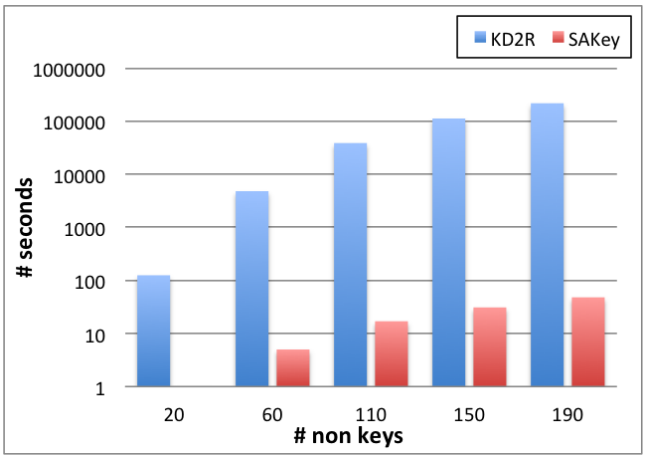


Fig. 4.16 KD2R vs. SAKey: Key derivation on *DB*:*BodyOfWater*

minute.

Additionally, to prove the efficiency of SAKey over KD2R, we compare their runtimes on seven classes of YAGO and DBpedia. In Table 4.8, we observe that SAKey outperforms KD2R in every case since it discovers fast the set of minimal keys. For example, in the case of the class *DB*:*NaturalPlace*, KD2R takes more than 2 days, while SAKey only 5 minutes.

As we notice in Table 4.8, there can exist thousands of keys for each class. In Table 4.9, we present some of the 1-almost keys discovered by SAKey for the classes *DB*:*NaturalPlace*, *DB*:*BodyOfWater* and *DB*:*Lake*.

Class	**# non keys**	**# keys**	**KD2R**	**SAKey**
DB:*Website*	9	99	1s	1s
YA:*Building*	15	40	1s	1s
YA:*SportsSeason*	22	188	2s	1s
DB:*Lake*	58	589	1min10s	1s
DB:*Mountain*	49	901	8min	1s
DB:*BodyOfWater*	220	3966	$>$ 1 day	66s
DB:*NaturalPlace*	302	7142	$>$ 2 days	5min

Table 4.8 Key derivation on different classes

We notice that both algorithms provided by SAKey to discover non keys and derive keys from non keys outperform the algorithms of KD2R in every dataset used in the experiments.

DB:Lake	***DB:BodyOfWater***	***DB:NaturalPlace***
{{*point*, *state*},	{*region*, *width*},	{{*range*, *flow*},
{*volume*, *point*},	{*originalStartPoint*},	{*river*, *length*, *city*},
{*mouthPosition*},	{*depth*, *region*},	{*volume*, *highestPlace*},
{*region*, *areaTotal*},	{*city*, *riverBranchOf*},	{*outflow*, *geology*},
{*riverMouth*, *inflow*},	{*district*, *inv-regionServed*},	{*locatedInArea*, *watershed*},
{*riverMouth*, *elevation*},	{*startPoint*, *geology*},	{*long*, *depth*, *name*},
...}	...}	...}

Table 4.9 1-almost keys for the classes *DB*:*NaturalPlace*, *DB*:*BodyOfWater* and *DB*:*Lake*

4.6.4 Data linking with *n*-almost keys

In this section, we evaluate the quality of identity links that can be found using *n*-almost keys. We have exploited two different datasets provided by the OAEI 2010 and OAEI 2013 for the instance matching track.

OAEI 2010. In the first experiment, we evaluate the quality of identity links, found between the datasets *D*3, *D*4 provided by OAEI 2010, first introduced in the Section 3.5.1.1. For clarity reasons, we remind that both datasets contain instances of the classes *Restaurant* and *Address*. Each restaurant is described using the datatype properties *name*, *phoneNumber*, *hasCategory* and the object property *hasAddress*. An address is described using the datatype properties *street*, *city* and the object property *hasAddress*. As we observe in Table 4.10, when 2 exceptions are allowed, SAKey discovers *phoneNumber* while when no exceptions are allowed, the 1-almost key {*phoneNumber*,*category*} is found. Note that the set of *n*-almost keys remains the same even when the *n* reaches up to 100. Thus, we link the instances of the class *Restaurant* using the 1-almost keys and the 2-almost keys. In order to use the key *hasAddress*, found in both cases, we link the addresses using the 1-almost key of *D*3 {*street*, *city*}. As we notice in Table 4.11, when the 2-almost keys are applied, the recall increases by 12% and reaches up to 99.1% while the precision stays also the same. We notice though that even if the property *phoneNumber* is not a key, it has a high linking power. Indeed, in this dataset, there exist two distinct restaurants, found in the same place, sharing phone numbers. In Table 4.12, we see that comparing the linking power of the 1-almost key {*phoneNumber*,*category*} and the 2-almost key {*phoneNumber*}, the use of phone number as a key increases the recall by 74% and it reaches up to 94.6% while the precision is reduced only by 1.4%.

OAEI 2013. In the first experiment, the benchmark contains one original file and five test cases. Each test case contains a file that should be linked with the original one. This ex-

n	***n*-almost keys**
0, 1	$\{\{name\}, \{hasAddress\}, \{phoneNumber, category\}\}$
2,...100	$\{\{name\}, \{hasAddress\}, \{phoneNumber\}\}$

Table 4.10 *n*-almost keys for the class *Restaurant* of OAEI 2010

# exceptions	**Recall**	**Precision**	**F-measure**
0, 1	87.5%	85.9%	86.7%
2,...100	99.1%	86%	92.11%

Table 4.11 Data linking for the class *Restaurant* of OAEI 2010 using equality

Almost keys	**Recall**	**Precision**	**F-measure**
$\{name\}$	75.8%	94.4%	84.1%
$\{address\}$	34.8%	75%	47.5%
$\{phoneNumber, category\}$	20.5%	95.8%	33.8%
$\{phoneNumber\}$	94.6%	94.6%	94.6%

Table 4.12 Linking power of almost keys found in *D*3

periment is conducted using the file from the first case. The original file contains DBpedia descriptions of persons and locations, while the test case file contains the same instances but with descriptions that have been modified. More precisely, values of 5 properties have been changed by randomly deleting/adding characters, by changing the date format, and/or by randomly changing integer values. Each file contains 1744 triples describing 430 instances. Each person can be described by the datatype properties *birthName*, *birthDate*, *comment*, *label* and the object properties *almaMater*, *award*, *birthPlace* and *doctoralAdvisor*. The property *almaMater* is used to describe a high school or a university from which an individual has graduated. Each location can be described by the datatype properties *populationTotal*, *label*, *motto* and the object property *isPartOf*. The property *motto* represents a short sentence that encapsulates the ideals of a location.

Each file contains 1744 triples describing 430 instances, using 11 properties. The second file is taken from the first test case. We have applied SAKey to discover *n*-almost keys in the test case file, where stop words have been eliminated. For example, words like $''of''$, $''the''$, $''at''$, $''restaurant''$ have been removed. The discovered *n*-almost keys have been used to link the data described in the two files. Two different scenarios have been executed. In the first scenario, only the string equality has been used by the linking tool in order to evaluate

# exceptions	Recall	Precision	F-measure
0, 1	25.6%	100%	41%
2, 3	47.6%	98.1%	64.2%
4, 5	47.9%	96.3%	63.9%
6, . . . , 16	48.1%	96.3%	64.1%
17	49.3%	82.8%	61.8%

Table 4.13 Data Linking in OAEI 2013 using equality

# exceptions	Recall	Precision	F-measure
0, 1	64.4%	92.3%	75.8%
2, 3	73.7%	90.8%	81.3%
4, 5	73.7%	90.8%	81.3%
6, . . . , 16	73.7%	90.8%	81.3%
17	74.4%	82.4%	78.2%

Table 4.14 Data Linking in OAEI 2013 using similarity measures

the quality of keys without considering the data heterogeneity. Thus, in this scenario, two resources are linked when they have common values for all the n-almost key properties. The recall, precision and F-measure of our linking results has been computed using the gold standard provided by OAEI.

Table 4.13 shows the evaluation of the linking process, in terms of recall, precision and F-measure, when all the discovered n-almost keys are applied and when n varies from 0 to 18. Unsurprisingly, the more exceptions are allowed, the more the recall increases and the precision decreases. Nevertheless, we observe that when two exceptions are allowed (i.e., 2-almost keys), the recall increases by 22%, while the precision decreases only by 1.9%. Moreover, we notice that in this case, the highest F-measure is obtained (62.4%). Indeed, by allowing two exceptions, SAKey discovers the properties *motto* and *birthDate* as 2-almost keys. Both properties have a high precision even if they are not keys in every case (they have few exceptions). Although the F-measure is increasing significantly when n-almost keys are applied, we notice that even in the best case the recall is not very high (less than 50%). For this reason, in the second scenario, the linking tool uses similarity measures to link the data. Table 4.14 presents the results of data linking in terms of recall, precision and F-measure when similarity measures are applied. In this case, the recall reaches up to 75%. We notice that the precision is, in general, lower when similarity measures are used than when only equality is used, since two instances are more easily considered as equal.

1-almost keys	**3-almost keys**	**5-almost keys**
$\{label\}$, $\{birthName\}$, $\{populationTotal\}$, $\{award, doctoralAdvisor\}$, $\{doctoralAdvisor, birthPlace\}$, $\{motto, isPartOf\}$, $\{award, birthDate\}$, $\{almaMater, birthDate\}$, $\{doctoralAdvisor, birthDate\}$	$\{label\}$ $\{birthName\}$, $\{populationTotal\}$, $\{award, doctoralAdvisor\}$, $\{doctoralAdvisor, birthPlace\}$, $\{motto\}$, $\{BirthDate\}$	$\{label\}$, $\{birthName\}$, $\{doctoralAdvisor\}$, $\{motto\}$, $\{BirthDate\}$
16-almost keys	**17-almost keys**	
$\{label\}$, $\{birthName\}$, $\{populationTotal\}$, $\{doctoralAdvisor\}$, $\{motto\}$, $\{BirthDate\}$, $\{award, almaMater, birthPlace\}$	$\{label\}$, $\{birthName\}$, $\{populationTotal\}$, $\{doctoralAdvisor\}$, $\{motto\}$, $\{BirthDate\}$, $\{award, birthPlace\}$, $\{award, almaMater\}$	

Table 4.15 Set of n-almost keys when n varies from 1 to 17

Table 4.15 presents the sets of n-almost keys discovered when different values of n are applied. When the same set of n-almost keys is obtained with different n values, the table contains only the n-almost keys for the biggest n. We observe that the number of minimal n-almost keys varies according to the n value for two reasons. First, the more the n increases the more the n-almost keys become general, since sets of properties considered as n-non keys may be considered as n-almost keys afterwards. For example, the 1-almost key $\{award,\ birthDate\}$ becomes $\{birthDate\}$ when 3 exceptions are allowed. Moreover, new n-almost keys can be added. For example, $\{award, almaMater, birthPlace\}$ is introduced when the n is set to 16.

The linking power of each n-almost key is presented in Table 4.16, in terms of recall, precision and F-measure. We observe that, the 1-almost key $\{birthName\}$ has 0% recall. This means that no link can be found using this key. This occurs since there exist only 8 descriptions in the dataset having this property. Comparing the two keys with no exceptions $\{award,\ birthDate\}$ and $\{almaMater,\ birthDate\}$ with the 4-almost key $\{birthDate\}$, we notice that the recall of $birthDate$ alone reaches up to 32% while in the both keys it was less

that 15%. The precision of $\{birthDate\}$ remains high (98.6%). Respectively, comparing $\{motto, isPartOf\}$ that is a key with no exceptions with the 4-almost key $\{motto\}$, we notice that the recall using the *motto* alone increases almost 10%, while the precision falls to less than 5%.

Almost keys	**Recall**	**Precision**	**F-measure**
$\{label\}$	8.3%	100%	15.4%
$\{birthName\}$	0%	-	-
$\{populationTotal\}$	0.2%	100%	0.4%
$\{award, doctoralAdvisor\}$	0.4%	100%	0.9%
$\{doctoralAdvisor, birthPlace\}$	5.1%	100%	9.7%
$\{doctoralAdvisor\}$	5.3%	85.1%	10%
$\{motto, isPartOf\}$	1.1%	100%	2.2%
$\{motto\}$	10.4%	95.7%	18.8%
$\{award, birthDate\}$	9.3%	100%	17%
$\{almaMater, birthDate\}$	15.3%	100%	26.6%
$\{doctoralAdvisor, birthDate\}$	4.8%	100%	9.3%
$\{BirthDate\}$	32.5%	98.6%	49%
$\{award, almaMater, birthPlace\}$	48.1%	96.3%	64.2%
$\{award, birthPlace\}$	0.5%	100%	0.9%
$\{award, almaMater\}$	8.1%	49.3%	13.97%

Table 4.16 Linking power of almost keys for the OAEI 2013

4.6.5 Conditional keys

In this experiment, we present the conditional keys that are discovered in the dataset $D11$ (see Section 3.5.1.5). In this dataset, no keys were discovered since two duplicate instances exist. We have chosen to construct conditions using the property *ina*:*aPourGenre*. This property describes the different categories of contents that exist in the dataset. Thus, the conditions are of the form *ina*:*aPourGenre* $=''$ *Documentaire*$''$, *ina*:*aPourGenre* $=''$ *Reportage*$''$, etc. In this dataset, the contents are grouped in 42 different categories.

The main idea of this experiment is to discover the conditional keys that can be found for every given category in the data. The results show that conditional keys can be discovered for all the categories except *"Débat"* which is the value of the duplicates for the property *aPourGenre*. The keys are shown in Table 4.17.

We observe that different conditional keys are discovered depending on the value of the property *ina*:*aPourGenre*. Different categories lead, in some cases, to the same

Condition	Discovered conditional keys
Sketch *Magazine* ...	{{*ina*:*TitreCollection*, *ina*:*Participant*, *ina*:*TitrePropreIntegrale*, *ina*:*DateDiffusion*}}
Interview *Serie* ...	{{*ina*:*TitreCollection*, *ina*:*Participant*, *ina*:*TitrePropreIntegrale*, *ina*:*Duree*, *ina*:*DateCreationNotice*, *ina*:*DateDiffusion*}}
Chronique *Extrait* ...	{{*ina*:*TitreCollection*, *ina*:*Participant*, *ina*:*TitrePropreIntegrale*, *ina*:*Duree*, *ina*:*DateCreationNotice*, *ina*:*DateDiffusion*}, {*ina*:*TitreCollection*, *ina*:*Participant*, *ina*:*Theme*, *ina*:*TitrePropreIntegrale*, *ina*:*Duree*, *ina*:*DateCreationNotice*}}
Reportage	{{*ina*:*TitreCollection*, *ina*:*Participant*, *ina*:*TitrePropreIntegrale*, *ina*:*Duree*, *ina*:*DateCreationNotice*}}

Table 4.17 Conditional keys for the dataset $D10$

set of keys. In this table, we provide only (at most) two categories that can lead to the same set of keys. The rest are omitted. For example, when the conditions $aPourGenre =''$ *Sketch*$''$ or $aPourGenre =''$ *Magazine*$''$ are given, the conditional key {{*ina*:*TitreCollection*, *ina*:*Participant*, *ina*:*TitrePropreIntegrale*, *ina*:*DateDiffusion*}} is found. Note that, some conditions lead to more specific conditional keys as in the case of the condition $aPourGenre =''$ *Interview*$''$. A qualitative analysis of the discovered conditional keys is necessary. To do so, an expert evaluation or a data linking evaluation should be conducted.

4.7 Conclusion

In this chapter, we have presented SAKey, an approach for discovering keys on large RDF data under the presence of errors and duplicates. Knowing that errors or duplicates may lead to the loss of keys, we have proposed SAKey, an approach that is capable of dealing with this kind of data. SAKey discovers n-almost keys, sets of properties that are not keys due to few exceptions in the data. For reasons of efficiency, SAKey discovers first the complete set of maximal (n-1)-non keys and then exploits them to derive the complete set of minimal n-almost keys. To be even more scalable, SAKey applies a series of filtering techniques in order to discard a part of the data that cannot lead to (n-1)-non keys. Moreover, SAKey uses a number of pruning strategies that allows to discover fast all the maximal (n-1)-non keys. Using property ordering and instance ordering heuristics, SAKey manages to discover n-non keys even faster. In contrast to KD2R, SAKey is able to scale in large datasets composed

of a big number of properties. Unlike the key derivation algorithm used in KD2R, SAKey introduces a new key derivation algorithm that is able to compute the sets of n-almost keys very efficiently. The key derivation is no longer the bottleneck of approaches that discover non keys first.

Our extensive experiments have showed that SAKey can run on millions of triples thanks to its filtering techniques and its pruning strategies. The scalability of the approach is validated on different datasets. Even when many exceptions are allowed, SAKey can still discover n-non keys very efficiently. Moreover, we observe that SAKey is much faster than KD2R both in discovering non keys and in deriving keys from non keys. Finally, the data linking experiment shows that the data linking results improve when few exceptions are allowed. Thus, the experiments demonstrate globally the validity and relevance of the discovered keys. A preliminary experiment to evaluate C-SAKey has been conducted and has demonstrated that conditional keys can be discovered in datasets where keys cannot be found.

Chapter 5

Conclusion and Perspectives

Enriching the knowledge in the Semantic Web is today a crucial point. In this thesis, we pursued this goal by studying, designing and implementing approaches capable of discovering OWL2 keys in RDF data. These RDF data can be numerous, incomplete and they can contain errors or duplicates. In the following, we first summarize the main achievements and then we discuss about future perspectives.

5.1 Thesis Summary

We have proposed two approaches, KD2R and SAKey, that are able to discover the complete set of OWL2 keys from RDF datasets, each conforming to an OWL ontology. These two approaches have been introduced to answer to the following problems:

How to discover keys in RDF data when the CWA cannot be asserted?
Theoretically, a key discovery approach cannot obtain meaningful results when the complete set of sameAs statements is not declared in the explored dataset. Therefore, in KD2R, we have exploited datasets for which the UNA is fulfilled. Moreover, in the setting of OWA, since not all the property instances are known, we have proposed two heuristics (see Chapter 3) to interpret the potential absence of information. More precisely, the pessimistic and optimistic heuristics have been introduced. In the pessimistic heuristic, when a property is not instantiated for one instance, all the values that exist in the dataset for this property are considered as possible values while, in the case of instantiated properties, we assume that the information is complete. In the optimistic heuristic, we assume that, if there exist values that are not known, they are different from all the values that already exist in the dataset. KD2R implements both the heuristics while SAKey only considers the optimistic

heuristic. The experiments have shown that the optimistic heuristic gives better linking results than the pessimistic heuristic.

How to discover keys in different datasets that conform to distinct ontologies?
In this situation, we have chosen to use a strategy in which keys are learned separately in each dataset. Then, mappings between classes and properties are considered in order to merge the discovered keys. The proposed merging operation computes keys that are valid in every dataset. The reason why we treat each dataset separately, is that UNA cannot be ensured when the datasets are seen as a single dataset. Both KD2R and SAKey apply this merging strategy. As seen in the experiments, keys obtained after this operation can be useful in the data linking process. However, we have not yet tested other strategies that could be applied to merge keys.

How to deal with duplicates or erroneous data in the key discovery process?
KD2R is not able to deal with data containing duplicates and errors. Therefore, we have proposed SAKey, an approach for key discovery when data are dirty. SAKey can discover sets of properties that are not keys due to few exceptions. To specify the number of allowed exceptions, a value n is fixed. This value represents the biggest number of exceptions allowed for this n-almost key. The experiments of SAKey have been conducted to compare SAKey to KD2R and to evaluate the effects of using almost keys in the data linking task. The experiments have proved that almost keys can have a very positive effect in the data linking, since sets of properties with high linking power are discovered and exploited. In this work, we have supposed that the value of n is fixed by an expert. Assigning an appropriate value for n is very important since, as shown in the experiments, when n is too high, poor quality keys can be discovered. However, it can be difficult for an expert to fix this value since it is related to the quality of the data.

We have also proposed an extension of SAKey, C-SAKey, an approach that discovers conditional keys. A conditional key is a key that is valid in subparts of the data that fulfill a specific condition. This condition is defined by specifying constant values for a set of properties. A preliminary experiment has shown that conditional keys can be discovered in datasets where no keys can be found. Indeed, when different values for the same condition are used, keys vary largely. Nevertheless, this extension does not include any additional pruning strategies and should be improved.

How to define a key discovery process that can scale?
To ensure the scalability of the key discovery process, we discover first maximal non keys and use them to derive minimal keys. Indeed, to decide that a set of properties is a non key, only a subpart of the data should be explored. Furthermore, prunings have been defined to speed up the discovery of non keys. In KD2R, an extended version of the prefix tree, proposed in [SBHR06], has been used to represent the data. This prefix tree can take into account both single valued and multivalued properties. KD2R applies several standard pruning strategies. Moreover, KD2R exploits class hierarchies, by using keys found in superclasses to prune the search space in subclasses. The experiments have demonstrated that KD2R can scale in datasets where the number of properties for each class is relatively small. In cases of large datasets, the structure used to store the data cannot be constructed. Note that, the optimistic heuristic can scale in bigger datasets than the pessimistic one.

To scale in large RDF datasets published on the Web, such as DBpedia and YAGO, a very compact data structure, called final map, has been proposed and used in SAKey. Moreover, we have introduced a series of filtering and pruning strategies to discover more efficiently the complete set of non keys. Some of these prunings are based on semantic dependencies that are discovered during the non key discovery process. Once the set of n-almost keys is discovered, a key derivation approach is needed. The process of deriving keys from non keys has been proven as the bottleneck of approaches that compute non keys first. Thus, in SAKey, we have introduced a new algorithm for deriving very efficiently, minimal n-almost keys by using the frequencies of properties in the complement sets. The experiments have highlighted that SAKey is orders of magnitude faster than KD2R in all the datasets used in the experiments and that it scales in large datasets that KD2R is not able to tackle. Furthermore, the efficiency of SAKey is not affected by the number of allowed exceptions. As demonstrated in the experiments, the new key derivation algorithm, introduced by SAKey, is much more efficient.

5.2 Future works

In this section, we outline various avenues for future work. We start by giving short-term future works and continue with more long-term future works.

Experimental evaluation of SAKey. To begin, we plan to conduct more experiments to evaluate more deeply the benefits of n-almost keys in the data linking problem.

Efficient algorithm for C-SAKey. An algorithm that discovers in an efficient way the complete set of conditional keys should be proposed. New pruning techniques should be also applied in this setting.

Graphical inteface for experts. Since n-almost keys represent probable keys, it would be interesting to create a graphical interface where an expert can see all the discovered n-almost keys and their exceptions, in order to decide whether these keys can be used or not to enrich the ontology.

Setting automatically the value of n. In the setting of n-almost keys, several extensions can be considered. In the current version of SAKey, the value n, representing the number of allowed exceptions in a n-almost key, is defined by an expert. Assigning a significant number to n can be very difficult for an expert, since it depends on the quality of the data. However, we have shown that, unsurprisingly, the quality of the discovered keys depends on this value. Allowing no exceptions might be very strict in RDF data, while allowing a huge number of exceptions might lead to many false negatives. Therefore, we are interested in proposing ways to automatically set the value of n, applying a preprocessing step on the data. Statistics on the data can be exploited, since measures such as the distribution of values can be used to discover an appropriate n.

Merging key strategies. Different strategies of merging sets of keys found in different datasets could be used. One strategy could be to define a "voting system" that prevents the loss of important keys that are valid in many datasets but not in all of them. For example, consider that the property $telephone$ is a minimal key for the class *Restaurant* in eight datasets, while in two other datasets the set of properties $\{telephone, name\}$ is a minimal key. In this case, the key $telephone$ can be seen as more probable to be correct since it is found in many datasets. Thus, "voting" can lead to avoid losing important keys. Similar strategies can be also applied in almost keys and conditional keys.

Construction of complex similarity functions using the value n. Considering that n represents a confidence about a discovered n-almost key, this value could be used to construct weighted similarity functions that can be exploited in the data linking process. Intuitively, almost keys with smaller n should have bigger impact in the linking process. For example, if the 3-almost key $\{DateOfBirth, LastName\}$ and 10-almost key $\{FirstName, LastName\}$ are discovered, a bigger weight should be assigned to the 3-almost key since it is more

probable to construct correct links.

Metric for the quality of keys. KD2R does not use any metrics to automatically evaluate the quality of the discovered keys. In SAKey, the value n allows to measure the number of exceptions for a given set of properties. The quality of both keys and n-almost keys could be measured using simple measures such as the support. Even if the support is a very simple measure, it appears to be very useful, since it allows avoiding keys that are valid only for few instances in the data.

Keys in heterogeneous RDF data. Both KD2R and SAKey consider that either the literal values are homogeneously represented or that a simple normalization step is applied before the discovery. In the future, we plan to extend both approaches in order to be able to discover keys, almost keys and conditional keys even when the data are heterogeneous. To do so, different similarity measures should be applied during the key discovery.

Keys involving chains of properties. In this thesis, the considered keys involve only properties related to one class and not chains of properties i.e., properties that are used in different classes. For example, in the Figure 5.1 that presents two distinct museums and their addresses, the set of properties $\{hasName, hasAddress/isInCity\}$ can be a key containing the property *hasName*, used for the class *Museum*, and the property *isInCity* of the class *Address*. Thus, keys, almost keys and conditional keys involving chains of properties could be discovered.

$hasName(museum1, "Louvre\ Museum"), hasAddress(museum1, address1),$
$hasName(museum2, "Louvre\ Museum"), hasAddress(museum2, address2),$
$isInCoutry(address1, "France"), isInCoutry(address2, "France")$
$isInCity(address1, "Paris"), isInCity(address2, "Lille")$

Fig. 5.1 RDF data for keys with chains

Discovery of semantic dependencies. It would be interesting to extract the semantic dependencies that are discovered during the n-non key discovery of SAKey. These semantic dependencies could be stored for example, to complete RDF descriptions of incomplete data.

Efficient update of keys when data evolve. Finally, we plan to study strategies that allow the update of keys when data change, without having to recompute the set of keys.

* * *

References

[ACC+14] Manuel Atencia, Michel Chein, Madalina Croitoru, Michel Leclere Jerome David, Nathalie Pernelle, Fatiha Saïs, Francois Scharffe, and Danai Symeonidou. Defining key semantics for the rdf datasets: Experiments and evaluations. In *Proceedings of the International Conferences on Conceptual Structures (ICCS))*, 2014.

[ADS12] Manuel Atencia, Jérôme David, and François Scharffe. Keys and pseudo-keys detection for web datasets cleansing and interlinking. In *EKAW*, pages 144–153, 2012.

[AN11] Ziawasch Abedjan and Felix Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 1565–1570, New York, NY, USA, 2011. ACM.

[BFG+07] P. Bohannon, Wenfei Fan, F. Geerts, Xibei Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 746–755, April 2007.

[BL06] Tim Berners-Lee. Linked data. *W3C Design Issues*, 2006.

[BLGS05] Mustafa Bilgic, Louis Licamele, Lise Getoor, and Ben Shneiderman. D-dupe: An interactive tool for entity resolution in social networks. In Patrick Healy and Nikola S. Nikolov, editors, *International Symposium on Graph Drawing*, volume 3843 of *Lecture Notes in Computer Science*, pages 505–507. Springer, September 2005.

[CCL+14] Michel Chein, Madalina Croitoru, Michel Leclere, Nathalie Pernelle, Fatiha Saïs, and Danai Symeonidou. Définition de la sémantique des clés dans le web sémantique: un point de vue théorique. In *Ingénierie des Connaissances (IC)*, 2014.

[CM08] Fei Chiang and Renée J. Miller. Discovering data quality rules. *Proc. VLDB Endow.*, 1(1):1166–1177, August 2008.

[Coh68] J Cohen. Weighted kappa: nominal scale agreement with provision for scaled disagreement or partial credit. *Psychological Bulletin*, 70(4):213–220, 1968.

[CRF03] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration*, pages 73–78, August 2003.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[EIV07] 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.

[FGLX11] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):683–698, 2011.

[FNS11] Alfio Ferrara, Andriy Nikolov, and François Scharffe. Data linking for the semantic web. *Int. J. Semantic Web Inf. Syst.*, 7(3):46–76, 2011.

[FS69] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64:1183–1210, 1969.

[FS99] Peter A. Flach and Iztok Savnik. Database dependency discovery: A machine learning approach. 1999.

[GHJ+10] Hector Gonzalez, Alon Y. Halevy, Christian S. Jensen, Anno Langen, Jayant Madhavan, Rebecca Shapley, Warren Shen, and Jonathan Goldberg-Kidon. Google fusion tables: web-centered data management and collaboration. In *SIGMOD Conference*, pages 1061–1066, 2010.

[GKK+08] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. On generating near-optimal tableaux for conditional functional dependencies. *Proc. VLDB Endow.*, 1(1):376–390, August 2008.

[GKM+03] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2):140–174, June 2003.

[GL96] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[GTHS13] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Amie: Association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 413–422, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[HCQ11] Wei Hu, Jianfeng Chen, and Yuzhong Qu. A self-training approach for resolving object coreference on the semantic web. In *WWW*, pages 87–96, 2011.

[HJAQR+13] A. Heise, Jorge-Arnulfo, Quiane-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *VLDB*, 7(4):301–312, 2013.

[HKPT99] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[IB12] Robert Isele and Christian Bizer. Learning expressive linkage rules using genetic programming. *PVLDB*, 5(11):1638–1649, 2012.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103, 1972.

[KLL13] Henning Köhler, Uwe Leck, and Sebastian Link. Possible and certain sql keys. Technical report, Centre for Discrete Mathematics and Theoretical Computer Science, 2013.

[LLTY13] Jiuyong Li, Jixue Liu, Hannu Toivonen, and Jianming Yong. Effective pruning for the discovery of conditional functional dependencies. *Comput. J.*, 56(3):378–392, 2013.

[LPL00] Stephane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl, and Torsten Grust, editors, *EDBT*, volume 1777 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2000.

[Mah97] M. J. Maher. Constrained dependencies. In *Theoretical Computer Science*, volume 1 of *Theoretical Computer Science*, pages 113–149, 1997.

[MR94] Heikki Mannila and Kari-Jouko Räihä. Algorithms for inferring functional dependencies from relations. *Data Knowl. Eng.*, 12(1):83–99, February 1994.

[NA11] Axel-Cyrille Ngonga Ngomo and Sören Auer. Limes a time-efficient approach for large-scale link discovery on the web of data. In *IJCAI*, pages 2312–2317, 2011.

[NC01] Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In Jan Van den Bussche and Victor Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2001.

[NdM12] Andriy Nikolov, Mathieu d'Aquin, and Enrico Motta. Unsupervised learning of link discovery configuration. In *9th Extended Semantic Web Conference (ESWC)*, pages 119–133, Berlin, Heidelberg, 2012. Springer-Verlag.

[NKAJ59] Howard B. Newcombe, James M. Kennedy, S.J. Axford, and A.P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, October 1959.

[NL12] Axel-Cyrille Ngonga Ngomo and Klaus Lyko. Eagle: Efficient active learning of link specifications using genetic programming. In *9th Extended Semantic Web Conference (ESWC)*, pages 149–163, 2012.

[PJ13] Shvaiko Pavel and Euzenat Jerome. Ontology matching: state of the art and future challenges. *IEEE Transactions on knowledge and data engineering*, 5(1):158–176, 2013.

[PSHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax Section 5. RDF-Compatible Model-Theoretic Semantics. Technical report, W3C, December 2004.

[PSS13] Nathalie Pernelle, Fatiha Saïs, and Danai Symeonidou. An automatic key discovery approach for data linking. *Journal of Web Semantics*, 23:16–30, 2013.

[QSSR12] Gianluca Quercini, Jochen Setz, Daniel Sonntag, and Chantal Reynaud. Facetted browsing on extracted fusion tables data for digital cities. In *proceedings of the Web of Linked Entities (WoLE) workshop in conjonction with ISWC 2012 conference*, pages 94–105, 2012.

[SAPS14] Danai Symeonidou, Vincent Armant, Nathalie Pernelle, and Fatiha Saïs. SAKey: Scalable Almost Key discovery in RDF data. In *Proceedings of the 13th International Semantic Web Conference (ISWC2014)*, ISWC '14. Springer Verlag, 2014.

[SAS11] Fabian M. Suchanek, Serge Abiteboul, and Pierre Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *The Proceedings of the VLDB Endowment(PVLDB)*, 5(3):157–168, 2011.

[SBHR06] 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 Very Large Data Bases (VLDB)*, VLDB '06, pages 691–702. VLDB Endowment, 2006.

[SF93] Iztok Savnik and Peter A. Flach. Bottom-up induction of functional dependencies from relations. In *In Proceedings of the AAAI'93 Workshop on Knowledge Discovery in Databases*, pages 174–185. AAAI Press, 1993.

[SH11] Dezhao Song and Jeff Heflin. Automatically generating data linkages using a domain-independent candidate selection approach. In *Proceedings of the 10th International Semantic Web Conference(ISWC) - Volume Part I*, ISWC'11, pages 649–664, Berlin, Heidelberg, 2011. Springer-Verlag.

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 697–706, 2007.

[SLZ09] Francois Scharffe, Yanbin Liu, and Chuguang Zhou. Rdf-ai: an architecture for rdf datasets matching, fusion and interlink. In *Proc. IJCAI 2009 workshop on Identity, reference, and knowledge representation (IR-KR), Pasadena (CA US)*, 2009.

[SNPR10] Fatiha Saïs, Nobal B. Niraula, Nathalie Pernelle, and Marie-Christine Rousset. LN2R a knowledge based reference reconciliation system: Oaei 2010 results. In *Proceedings of the 5th International Workshop on Ontology Matching (OM-2010)*, 2010.

[SPR09] Fatiha Saïs, Nathalie Pernelle, and Marie-Christine Rousset. Combining a logical and a numerical method for data reconciliation. *Journal on Data Semantics*, 12:66–94, 2009.

[SPS11] Danai Symeonidou, Nathalie Pernelle, and Fatiha Saïs. KD2R : a Key Discovery method for semantic Reference Reconcilation. In *7th International IFIP Workshop on Semantic Web & Web Semantics (SWWS 2011)*, LNCS, pages 392–401, Heraklion, Greece, October 2011. Springer Verlag.

[SS96] H. Saiedian and T. Spencer. An efficient algorithm to compute the candidate keys of a relational database schema. *The Computer Journal*, 39(2):124–132, 1996.

[VBGK09] Julius Volz, Christian Bizer, Martin Gaedke, and Georgi Kobilarov. Discovering and maintaining links on the web of data. In *Proceedings of the 8th International Semantic Web Conference(ISWC)*, ISWC '09, pages 650–665, Berlin, Heidelberg, 2009. Springer-Verlag.

[VLM12] S. Link V. Le and M. Memari. Schema- and data-driven discovery of sql keys. *JCSE*, 6(3):193–206, 2012.

[WDS+09] 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 *WebDB*, 2009.

[WGR01] Catharine Wyss, Chris Giannella, and Edward Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *In To appear in Lecture Notes in Computer Science (Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, pages 101–110, 2001.

[Yan04] Guizhen Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In Won Kim, Ron Kohavi, Johannes Gehrke, and William DuMouchel, editors, *KDD*, pages 344–353. ACM, 2004.

[YHB02] Hong Yao, Howard J. Hamilton, and Cory J. Butz. Fd_mine: Discovering functional dependencies in a database using equivalences. In *ICDM*, pages 729–732. IEEE Computer Society, 2002.