Découverte de schéma pour les données du Web sémantique

Kenza Kellou, Kenza Kellou-Menouer

NNT : -

THÈSE DE DOCTORAT
DE
L'UNIVERSITE PARIS-SACLAY
PRÉPARÉE A
L'UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN En YVELINES

École Doctorale $N°580$
Sciences et Technologies de l'information et de la communication (STIC)

Spécialité de doctorat : Informatique

par

Kenza Kellou-Menouer

Découverte de schéma pour les données du Web sémantique

Thèse présentée et soutenue publiquement à Versailles, le -

Composition du Jury :

M^{me}.	Comyn-Wattiau Isabelle	Professeur, CNAM	Rapporteur
M.	Mohand-Saïd Hacid	Professeur, Université Lyon 1	Rapporteur
M.	Ait-Ameur Yamine	Professeur, Université INPT-ENSEEIHT	Examinateur
M.	Masseglia Florent	MCF-HDR, Université Montpellier 2	Examinateur
M^{me}.	Sais Fatiha	MCF, Université Paris-Sud	Examinateur
M^{me}.	Kedad Zoubida	MCF-HDR, Université de Versailles	Directrice de thèse

Remerciements

C'est un grand honneur pour moi d'adresser mes respectueux remerciements à **Isabelle Comyn-Wattiau** et **Mohand-Saïd Hacid** pour avoir accepté d'être les rapporteurs de cette thèse et pour le temps qu'ils auront bien voulu y consacrer.

Mes vifs remerciements vont également à **Yamine Ait-Ameur**, **Florent Masseglia** et **Fatiha Sais** pour l'honneur qu'ils me font en acceptant de participer au jury de thèse.

J'adresse ma profonde gratitude à ma directrice de thèse **Zoubida Kedad** pour son soutien, ses remarques éclairées et son encadrement tout au long de cette thèse. Ses conseils étaient très précieux et allaient bien au-delà de l'obtention d'un titre universitaire. Ce fut un réel plaisir de travailler ensemble !

Je tiens à remercier tous les membres de l'équipe ADAM pour avoir été de bons collègues et pour leur attitude amicale ainsi que tous les membres du laboratoire DAVID pour leur accueil. Mes remerciements vont également à Chantal Ducoin et les autres secrétaires du laboratoire pour leur efficacité.

Je remercie mes chers parents pour leur soutien, leur aide et leur amour. Je remercie également mon frère pour ses conseils et ses encouragements.

Mes remerciements vont également à ma belle famille pour leur soutien et leurs encouragements.

Je tiens également à remercier mon époux pour son accompagnement, son soutien et sa patience, notamment durant la période de rédaction de ce manuscrit. J'ai également une pensée toute particulière pour notre petit Samy qui égaye notre vie tous les jours.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Découverte de schéma pour les données du Web sémantique

Mots clés : Données RDF(S)/OWL, Clustering, Annotation, Découverte de patterns.

Résumé : Un nombre croissant de sources de données interconnectées sont publiées sur le Web. Cependant, leur schéma peut être incomplet ou absent. De plus, les données ne sont pas nécessairement conformes au schéma déclaré. Ce qui rend leur exploitation complexe.

Dans cette thèse, nous proposons une approche d'extraction automatique et incrémentale du schéma d'une source à partir de la structure implicite de ses données. Afin de compléter la description des types découverts, nous proposons également une approche de découverte des patterns structurels d'un type. L'approche procède en ligne sans avoir à télécharger ou à parcourir la source. Ce qui peut être coûteuse voire impossible car les sources sont interrogées à distances et peuvent imposer des contraintes d'accès, notamment en termes de temps ou de nombre de requêtes.

Nous avons abordé le problème de l'annotation afin de trouver pour chaque type un ensemble de labels permettant de rendre compte de son sens. Nous avons proposé des algorithmes d'annotation qui retrouvent le sens d'un type en utilisant des sources de données de références. Cette approche s'applique aussi bien pour trouver des noms pertinents aux types découverts que pour enrichir la description des types existants.

Enfin, nous nous sommes intéressés à caractériser la conformité entre les données d'une source et le schéma qui les décrit. Nous avons proposé une approche pour l'analyse et l'amélioration de cette conformité et nous avons proposé des facteurs de qualité, les métriques associées, ainsi qu'une extension du schéma permettant de refléter l'hétérogénéité entre les instances d'un type.

Title : Schema Discovery in Semantic Web Data Sources

Keywords : RDF(S)/OWL Data, Clustering, Annotation, Patterns Discovery

Abstract: An increasing number of linked data sources are published on the Web. However, their schema may be incomplete or missing. In addition, data do not necessarily follow their schema. This flexibility for describing the data eases their evolution, but makes their exploitation more complex.

In our work, we have proposed an automatic and incremental approach enabling schema discovery from the implicit structure of the data. To complement the description of the types in a schema, we have also proposed an approach for finding the possible versions for each of them. It proceeds online without having to download or browse the source. This can be expensive or even impossible because the sources may have some access limitations, either on the query execution time, or on the number of queries.

We have also addressed the problem of annotating the types in a schema, which consists in finding a set of labels capturing their meaning. We have proposed annotation algorithms which provide meaningful labels using external knowledge bases. Our approach can be used to find meaningful type labels during schema discovery, and also to enrich the description of existing types.

Finally, we have proposed an approach to evaluate the gap between a data source and its schema. To this end, we have proposed a set of quality factors and the associated metrics, as well as a schema extension allowing to reflect the heterogeneity among instances of the same type. Both factors and schema extension are used to analyze and improve the conformity between a schema and the instances it describes

Table des matières

Table des figures

Liste des tableaux

List of Algorithms

Chapitre 1

Introduction générale

1.1 Contexte

Le Web a changé notre façon de communiquer, la façon dont nous menons nos achats et notre façon de rechercher des informations. Le Web sémantique se définit, selon, Tim Berners-Lee [14] comme « un Web de données, qui peuvent être traitées directement et indirectement par des machines », et ceci représente une immense base de données globalement liées.

Une quantité croissante de données sont disponibles sur le Web ; ces données sont décrites par les langages proposées par le W3C, comme RDF (Resource Description Framework) [15], RDFS (Ressource Description Framework Schema) [16] et OWL (Web Ontology Language) [17]. Ces langages permettent une description flexible des données, car ils n'imposent pas de structure stricte à laquelle elles doivent se conformer, contrairement aux bases de données classiques comme les bases de données relationnelles. Les données au format RDF sont décrites sous la forme d'un triplet $<$ sujet, prédicat, objet $>$. Le sujet reflète un type (classe/concept), le prédicat correspond à une relation (propriété) et l'objet reflète un autre type ou correspond à un littéral ; ce qui permet facilement d'indiquer les relations entre les types ou littéraux. Par exemple, dans « Picasso peint Guernica », le sujet est « Picasso », le prédicat est « peint » et l'objet est « Guernica ». RDF permet également de spécifier le type des ressources comme suit : « Picasso *rdf*:*type* Peintre », « Guernica *rdf*:*type* Œuvre ». RDFS est fondé sur RDF, et permet de définir des informations sur le schéma des données. Par exemple, il permet de spécifier que la propriété « peint » prend comme valeur de sujet un *Peintre* à travers la déclaration « peint *rdfs*:*domain* Peintre », et qu'elle prend comme valeur d'objet une *Œuvre* à travers la déclaration « peint *rdfs*:*range* Œuvre ». Elle permet également de spécifier les liens de hiérarchies entre les concepts, comme pour dire qu'un artiste est une généralisation d'un peintre, à travers la déclaration suivante : « Peintre $rdfs$:$subClassOf$ Artiste ». Enfin, OWL étend les possibilités de RDFS et permet de décrire des vocabulaires extrêmement riches : on parle alors d'ontologies.

Les jeux de données liées sont publiés suivant un ensemble de principes [18]

tels que l'utilisation des URI comme noms pour les objets, et la définition de liens vers d'autres sources de données. La figure 1.1 est un exemple d'un jeu de données liées. Les nœuds sont connectés à l'aide de liens représentant des propriétés. Un nœud peut représenter soit un type/classe tel que « Artiste » dans notre exemple, soit une instance liée à son type par la propriété *rdf:type*, comme « Picasso », ou un littéral. Comme on peut le voir dans l'exemple, un jeu de données peut inclure des informations liées au schéma, qui pourraient être un type défini dans le jeu de données tel que « Artiste » et des informations relatives à l'instance, comme les propriétés de l'instance « Picasso ».

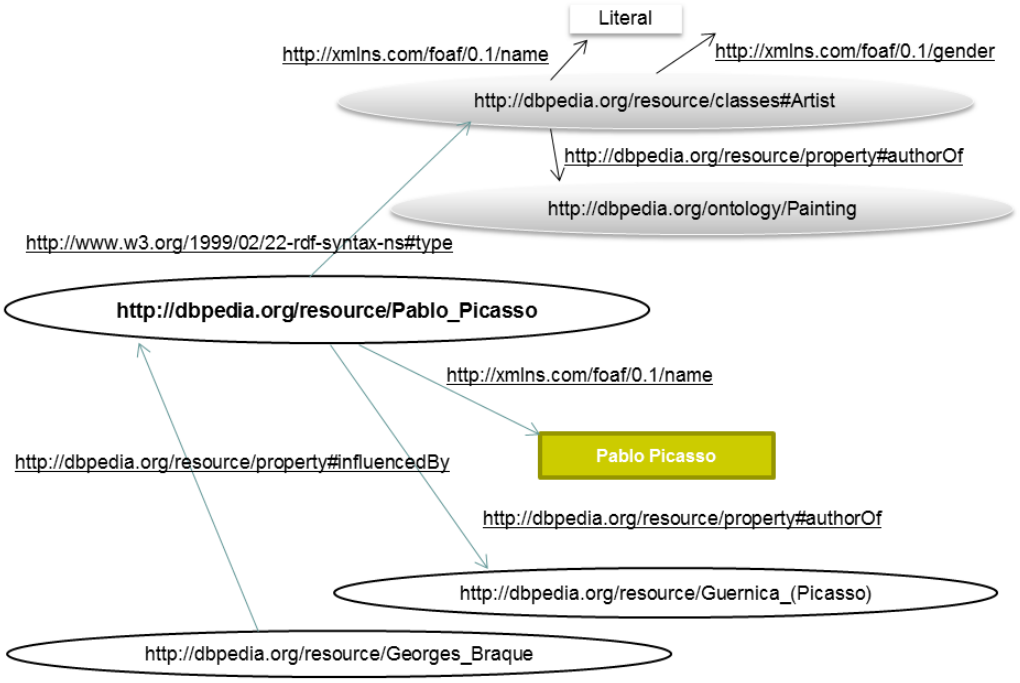


FIGURE 1.1 – Exemple de données liées.

Afin de faciliter l'exploitation des sources de données du Web sémantique et de résoudre l'hétérogénéité terminologique qu'elles peuvent présenter, il est recommandé lors de la création d'un jeu de données d'utiliser autant que possible des vocabulaires existant. En effet, plusieurs vocabulaires standards ont été proposés, comme *FOAF* qui décrit les relations entre amis ou encore *GN* qui décrit les données géographiques et cartographiques. Ces vocabulaires sont disponibles sur le *Linked Open Vocabularies (LOV)* [19, 13] qui regroupe 520 vocabulaires publiés par des organismes et des individus couvrant un large éventail de domaines.

Le *Linked Open Data Cloud (LOD Cloud)* [20] est un ensemble de jeux de données RDF qui ont été publiés sur le Web et qui sont interconnectés entre eux. Un exemple remarquable pour une source de données ouverte liée est *DBpedia*, qui représente la plus importante source de données RDF actuelle pour

le Web sémantique. C'est un projet communautaire permettant d'extraire les articles contenus sur *Wikipédia* sous forme de triplets RDF, afin de rendre ces informations utilisables par les applications.

Dans les sources de données du Web sémantique, les déclarations sur le schéma des données sont fournies avec les données, et une source décrite en RDF(S)/OWL contient aussi bien des instances et leurs propriétés que les types auxquels ces instances appartiennent. Mais les déclarations relatives au schéma peuvent être manquantes ou inexistantes. En effet, les langages utilisés pour décrire ces données n'imposent pas que les données soient conformes aux déclarations sur le schéma ; ainsi, une instance d'un type peut être décrite par des propriétés non déclarées pour son type, et les instances d'un même type peuvent également être décrites par des ensembles de propriétés hétérogènes. Cette très grande flexibilité des langages offre une grande souplesse de description ; ce qui favorise l'évolution des données, mais complique leur exploitation. Par exemple, il est difficile de poser une requête sur une source de données sans connaître son schéma, en termes de types et de propriétés.

L'adoption croissante des principes des données liées a conduit à la disponibilité d'un grand nombre de jeux de données sur le Web. Le Web est devenu un espace d'information formé de sources de données interconnectés permettant la conception d'applications innovantes. Cependant, l'utilisation de ces sources de données est entravée par le manque d'informations descriptives sur leur contenu. En effet, l'interconnexion et l'interrogation de ces jeux de données requièrent certaines connaissances sur leur schéma qui n'est souvent pas disponible.

Dans ce travail de thèse, nous nous sommes intéressés à proposer des techniques et des algorithmes qui permettent d'extraire des informations sur le contenu d'une source de données du Web sémantique, et notre objectif est de fournir une description d'une source à travers un résumé structurel de son contenu.

1.2 Problématiques

Obtenir une vue d'ensemble d'un grand jeu de données RDF et caractériser son contenu est souvent difficile. La compréhension d'un jeu de données devient encore plus difficile lorsque les informations relatives au schéma sont manquantes. Dans ces sources, les données ne sont pas nécessairement conformes aux déclarations relatives au schéma présentes dans la source de données : une instance d'un type peut être décrite par des propriétés non déclarées pour son type, comme elle peut ne pas être décrite par des propriétés déclarées pour son type. Cela offre une grande flexibilité dans la description des données, mais rend leur exploitation difficile.

Pour pouvoir produire un résumé structurel décrivant le contenu d'une source de données, nous avons identifié et abordé les problématiques suivantes :

Découverte de la structure implicite d'une source de données. La découverte de la structure implicite d'une source se base sur la structure implicite des données. La caractérisation de la source par un schéma permet de connaître son contenu en termes de types et les propriétés de chaque type. Ce schéma sera utilisé, par exemple, pour l'interrogation de la source de données. En effet, la formulation d'une requête est impossible sans connaître le contenu d'une source de données, en termes de types et de liens entre eux. Le schéma est également essentiel pour l'exécution de requêtes distribuées. Lorsqu'une requête est exécutée sur plusieurs sources de données, la disponibilité du schéma de chaque source permet de décomposer la requête et d'envoyer les sous-requêtes aux sources pertinentes [21]. Pour faire émerger la structure implicite des instances contenues dans une source, il faut les regrouper, et ceci pose plusieurs questions : quels sont les bons critères de regroupement qui permettent de gérer l'hétérogénéité des données, inévitable dans notre contexte ? quelle technique automatique répond aux caractéristiques des données liées, notamment le typage multiple d'une instance, l'évolution des données, l'existence de bruit dans les données ? Comment faire le regroupement sans connaître le nombre de types *a priori*, ou encore comment fixer la valeur du seuil de similarité ?

La découverte d'un schéma décrivant les types et les propriétés des instances, bien que nécessaire pour l'exploitation pertinente de la source, n'est pas suffisante : le schéma ne rend pas compte de la co-occurrence entre les propriétés d'un type. Ainsi, si des personnes sont caractérisées par un nom, une adresse et un mail dans une source de données, il est possible qu'aucune personne ne soit décrite à la fois par une adresse et un mail en même temps. La co-occurrence de certaines propriétés d'un type est importante, notamment pour l'interrogation. Dans le cas d'une source de données distante, plusieurs problèmes se posent. D'abord le parcours des données n'est pas possible, et le seul accès possible se fait au moyen de requêtes via un point d'accès SPARQL. De plus, le serveur de la source peut imposer des restrictions pour garantir à tous les mêmes chances d'interroger la source ; il peut s'agir de limitations sur le temps d'exécution d'une requête, la taille du résultat ou le nombre de requêtes envoyées. Pour retrouver les versions structurelles d'un type, il faut énumérer les différentes versions possibles ; ce qui pose un problème combinatoire lorsque le nombre de propriétés du type est grand.

Enrichissement d'une structure par des informations qui capturent son sens. En plus du schéma d'une source de données, il est utile d'être en mesure de rendre compte du sens, à la fois des types découverts ou de ceux qui sont déclarés dans la source. En effet, certaines déclarations peuvent ne pas capturer par leur terminologie tout le sens des instances d'un type. Par exemple, pour un ensemble d'instances ayant le type *Politicien*, comment savoir s'il s'agit de ministres ? de présidents ? ou encore de savoir quel est le pays dont il s'agit ?

Ces informations ne sont pas forcément présentes dans la source de données.

Le défi qui est ici posé est comment combler ce manque de sémantique ? Où trouver les informations qui "expliquent" ces données ? Lorsque plusieurs informations sont retrouvées, comment évaluer leur pertinence relative ?

Évaluation de la distance entre le schéma et les données d'une source. Dans ce contexte de très forte hétérogénéité, et si le schéma décrivant une source de données est disponible, une question qui se pose est de savoir quelle est la distance entre les données et le schéma qui les décrit ? En effet, une instance peut être décrite par des propriétés qui ne sont pas déclarées pour son type ; et elle peut également ne pas être décrite par toutes les propriétés déclarées pour son type. Il est alors important de pouvoir caractériser l'écart existant entre le schéma et les données avant d'exploiter la source : exploiter un schéma qui est très éloigné des instances qu'il est censé décrire n'est pas utile. Cela présente plusieurs interrogations : quelles sont les différentes facettes de la conformité entre un schéma et les données qu'il décrit ? Comment évaluer chacune de ces facettes ? et comment réduire l'écart existant ?

Ces différentes problématiques visent à trouver des solutions pour caractériser le contenu d'une source de données faiblement structurée et hétérogène. Nous présentons dans la section qui suit nos différentes contributions pour répondre à ces problématiques.

1.3 Contributions

Dans cette thèse nous proposons différentes contributions relatives à l'extraction du schéma d'une source de données RDF :

Découverte du schéma implicite des données [ER2015, TLDKS2016, ESWC2015, EGC2105, FDC2015]. Nous proposons de découvrir le schéma d'une source à partir de la structure implicite de ses données ; ce schéma est composé de types et de liens entre eux. Pour découvrir les types, nous avons proposé une approche qui est une extension de l'algorithme de clustering DBscan. L'approche proposée permet de détecter automatiquement le seuil de similarité pour le regroupement des instances. Elle permet également d'attribuer plusieurs types à une instance ainsi qu'un typage incrémental des instances. Un profil probabiliste qui caractérise chaque type est construit lors du processus de découverte des types. Ce profil est constitué de l'ensemble des propriétés utilisées pour décrire les instances du type, et à chaque propriété est associée une probabilité. Nous proposons également de découvrir deux types de liens entre les types. Des liens sémantiques qui décrivent le domaine et le co-domaine des propriétés, et des liens hiérarchiques qui décrivent les liens de généralisation entre les types. Les

liens sémantiques sont découverts en analysant les profils des types, et les liens hiérarchiques en utilisant un algorithme de clustering hiérarchique sur les profils de types.

Découverte des versions structurelles des types [SSDBM2017]. Comme les instances d'un même type peuvent être décrites par des propriétés différentes, nous proposons de caractériser un type par les différents patterns structurels (versions) de ses instances pour renseigner la co-occurrence entre ses propriétés. Nous proposons *SchemaDecrypt*, une approche qui permet de découvrir les versions d'un type dans une source de données distante, interrogée au moyen de requêtes posées depuis un point d'accès SPARQL. SchemaDecrypt parvient à surmonter l'explosion combinatoire du nombre de versions candidates et de découvrir les versions en ligne, sans télécharger ni parcourir la source de données. Pour trouver les différentes versions d'un type, nous utilisons un profil de type probabiliste pour guider l'exploration des versions candidates. Nous réduisons le nombre de versions candidates en découvrant des règles d'inclusion et d'exclusion entre les propriétés d'un type. Nous avons également proposé une exploration parallèle des versions avec *SchemaDecrypt* + + qui améliore considérablement la performance. En effet, la présence de règles d'exclusion permet de former des modèles de versions explorables en parallèle.

Les versions d'un type permettent de caractériser la source de données avec un degré de précision plus important, en décrivant la co-occurrence entre les propriétés et leur nombre d'occurrences. Ces versions peuvent être utiles par exemple pour la décomposition de requêtes sur des sources de données distribuées. En effet, les versions d'un type résument les structures de ses instances dans chaque source, ce qui est utile pour décomposer et formuler les sous-requêtes pour obtenir la réponse la plus complète possible. En outre, le nombre d'occurrences de chaque version permet d'optimiser et d'estimer le coût d'un plan d'exécution.

Annotation des données [OTM2016]. Pour trouver le sens de chaque type découvert, nous proposons une approche qui permet d'annoter une instance ou un groupe d'instances qui représentent un type. Nous proposons d'extraire à partir de bases de connaissances disponibles sur le Web, un ensemble de termes (annotations) pondérés selon leur pertinence et qui expliquent ce que contient un groupe (cluster ou classe) d'entités. Dans le cas où le schéma est fourni explicitement dans le jeu de données, l'annotation permettrait également de mieux expliquer des types en les enrichissant par des annotations complémentaires. En effet, même si le type est déclaré, son nom ne reflète pas nécessairement le sens de son contenu. L'annotation permet de renseigner le sens des types, ce qui pourrait avoir plusieurs utilités. Les annotations découvertes peuvent être exploitées pour l'alignement de différents jeux de données, et ce, pour

surmonter leur hétérogénéité. Les types ayant la même sémantique pourraient être identifiés en comparant leurs ensembles respectifs d'annotations. De la même manière, les annotations découvertes peuvent également être exploitées à des fins d'interconnexion entre les jeux de données.

Analyse et amélioration de la conformité entre une source de données RDF et son schéma [QMMQ2015]. Nous proposons un ensemble de facteurs et de métriques associées qui permettent d'évaluer la conformité entre une source de données RDF et son schéma. Nous proposons également une extension du schéma qui permet de mieux refléter l'hétérogénéité des données. Comme les données peuvent évoluer sans respecter le schéma déclaré, nous proposons également une approche qui permet de répercuter sur le schéma les changements survenant au niveau des données.

1.4 Organisation du manuscrit

Ce manuscrit de thèse est organisé de la façon suivante :

Dans le chapitre 2, nous présentons dans un état de l'art les différentes familles d'approches relatives à l'extraction du schéma, et nous avons pour cela identifié les catégories suivantes : (i) approches de découverte du schéma implicite ; (ii) approches d'enrichissement du schéma explicite ; (iii) approches de découverte des patterns structurels des instances ; et (iv) approches d'annotation des données. Les approches sont décrites et comparées entre elles, puis une analyse de leurs limites par rapport aux besoins des données du Web sémantique est présentée.

Dans le chapitre 3, nous proposons une approche automatique de découverte du schéma implicite d'un graphe de données RDF, avec : (i) la détection automatique du paramètre du seuil de similarité ; (ii) la construction d'un profil probabiliste pour chaque type découvert ; (iii) l'attribution de types multiples à une entité et (vi) le typage incrémental d'une entité nouvelle.

Dans le chapitre 4, nous proposons *SchemaDecrypt* qui est une approche de découverte des patterns structurels exacts des types d'une source de données distante. En effet, les versions d'un type permettent de caractériser la source de données avec un degré de précision plus important. Cette approche doit faire face essentiellement à un problème combinatoire ainsi qu'aux restrictions imposées par les sources de données. Nous proposons également d'améliorer les performances de cette approche par une exploration parallèle des versions potentielles avec *SchemaDecrypt* + +.

Dans le chapitre 5, nous proposons un ensemble d'algorithmes d'annotation dédiés aux données du Web sémantiques, qui permettent de capturer la sémantique des groupes d'entités à partir de sources de connaissances externes disponibles sur le Web. Le but de cette proposition est notamment de pouvoir trouver le nom des types découverts par l'approche proposée dans le chapitre 1, mais également d'améliorer la compréhension des types déclarés.

Nous abordons dans le chapitre 6, l'évaluation de la conformité entre un graphe de données RDF et son schéma. Nous proposons également une extension d'un schéma exprimé en RDF qui permet de réduire l'écart existant entre le schéma et les données correspondantes, et également de refléter l'évolution des données.

Enfin, nous concluons ce manuscrit par un bilan des travaux effectués et nous présentons quelques perspectives de recherche dans le chapitre 7.

Chapitre 2

État de l'art

2.1 Introduction

Nous assistons aujourd'hui à la prolifération de sources de données faiblement structurées, irrégulières, incomplètes et massives ; c'est notamment le cas des données du Web sémantique, exprimées dans des langages, tel que RDF. Contrairement aux données structurées, ces sources ne suivent pas un schéma prédéfini ; ce qui engendre le besoin de caractériser leur contenu et de les « comprendre » pour pouvoir les utiliser de façon pertinente.

Dans notre travail, nous nous intéressons à l'extraction d'informations relatives au schéma pour des sources de données pour lesquelles ce schéma est absent ou encore partiellement défini. Ce problème a été identifié, comme l'une des directions de recherche des plus importantes pour la gestion de données [22].

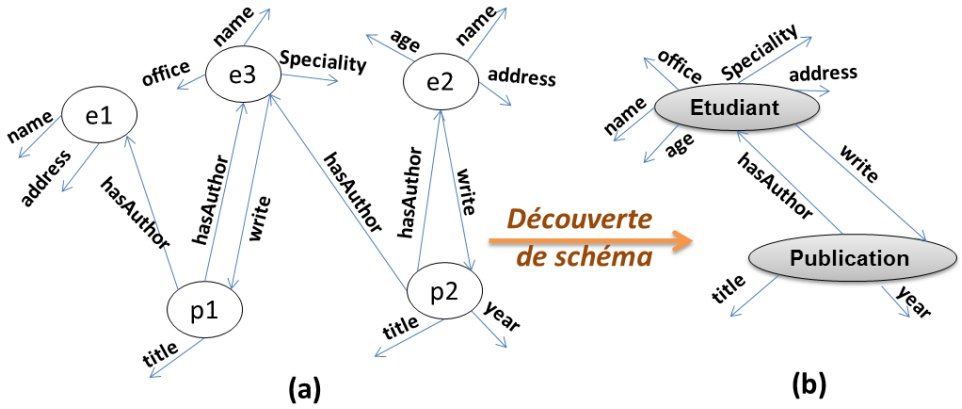


FIGURE 2.1 – Exemple d'un graphe de données (a) et de son schéma (b)

La figure 2.1 (a) montre un exemple de jeu de données liées. La découverte du schéma pour ce jeu de données consiste principalement en la découverte des types (classes) et des liens entre eux comme dans la figure 2.1 (b). En plus, de la découverte de la structure inhérente du jeu de données, il est également important de rendre compte du sens de chaque structure découverte. Un schéma de la source

de données qui décrit les types des données et les liens entre eux permet d'avoir une caractérisation du contenu de cette source de données, mais il est également important, en plus de connaître la structure implicite des données, de fournir la sémantique des structures découvertes.

Des approches ont été proposées pour la découverte de schéma. Certaines traitent les données du Web sémantique, d'autres ont été proposées pour des données graphes de type XML ou OEM. En plus de la découverte de la structure des données, certaines approches ont abordé le problème de l'annotation des structures découvertes pour capturer leur sémantique. Ce problème d'annotation a également été traité dans d'autres contextes, par exemple, celui des tables Web. Ces dernières sont des données structurées, il n'est donc pas nécessaire d'extraire leur schéma. Cependant, elles sont rarement explicites et les lignes d'en-tête ne sont pas toujours présentes. Il est par conséquent très utile de les annoter pour capturer leur sens.

Nous nous sommes intéressés dans cet état de l'art à identifier les approches qui permettent d'inférer des informations sur le contenu d'une source de données en terme de schéma, c'est à dire, les classes (types) et les liens entre ces classes. Certaines de ces approches découvrent le schéma d'un jeu de données à travers la structure implicite des données, sans faire l'hypothèse que certaines déclarations explicites sur le schéma sont fournies dans le jeu de données. Tandis que d'autres utilisent les déclarations explicites sur le schéma contenues dans le jeu de données afin de les compléter et de les enrichir. D'autres approches se sont intéressées à la découverte des patterns structurels contenus dans un jeu de données. Ces patterns ne représentent pas forcément des structures de classes mais une version structurelle possible des instances du jeu de données. Afin de caractériser et de capturer le sens des structures découvertes, nous nous sommes intéressés aux approches d'annotation des données. Parmi ces approches, certaines adressent l'annotation des tables Web. En effet, une table Web a une structure bien définie, cependant, sa sémantique peut être ambiguë à travers les en-têtes des colonnes et parfois même totalement absente.

Il existe d'autres approches qui se sont intéressées à décrire ces sources de données par profilage, comme dans les projets *ProLOD++* [23] ou *Metanome* [24]. Le profilage d'une source de données consiste généralement à extraire des informations sur la source de données, telles que le nombre de types dans la source de données, le nombre d'instances pour chaque type, le nombre d'instances ayant une propriété donnée, etc. Ces approches ne sont pas abordées dans ce chapitre car bien qu'elles découvrent des éléments liés au schéma (contraintes, dépendances fonctionnelles, etc), nous avons accès notre travail sur les types et les liens entre eux.

Dans la section 2.2, nous présentons les approches de découverte du schéma implicite d'un jeu de données. Ces approches découvrent le schéma d'un jeu de données à partir de la structure implicite des données. Cela, sans faire l'hypothèse

de l'existence de déclarations sur le schéma dans le jeu de données. Ces approches peuvent être classées en deux catégories : les approches qui découvrent le schéma implicite en regroupant les instances sur la base de leur similarité structurelle, qui sont présentées dans la section 2.2.1 et les approches qui regroupent les chemins similaires d'un jeu de données, présentées dans la section 2.2.2. Une discussion sur ces approches de découverte du schéma implicite d'un jeu de données est présentée dans la section 2.2.3.

Dans la section 2.3, nous présentons les approches d'enrichissement du schéma explicite du jeu de données. Le schéma explicite est composé de toutes les déclarations sur le schéma fournies dans le jeu de données. Ces approches utilisent ces déclarations sur le schéma afin de les compléter et de les enrichir. Contrairement aux approches présentées dans la section 2.2, ces approches requièrent certaines déclarations sur le schéma pour en inférer d'autres. Ces approches procèdent de différentes manières : certaines utilisent des techniques de fouille de données, elles sont présentées dans la section 2.3.1 et d'autres utilisent l'analyse statistique, elles sont présentées dans la section 2.3.2. Une discussion sur ces approches d'enrichissement du schéma existant est présentée dans la section 2.3.3.

Dans la section 2.4, nous présentons les approches de découverte des patterns structurels des données. Un pattern structurel est une version structurelle possible des instances d'un jeu de données. Celui-ci peut être exact ou approximatif avec certaines propriétés optionnelles. Les approches de découverte de patterns peuvent être classées selon qu'elles retrouvent des patterns exacts ou approximatifs : les approches qui découvrent des patterns structurels exacts sont présentées dans la section 2.4.1 ; les approches qui découvrent des patterns structurels approximatifs sont présentées dans la section 2.4.2. Une discussion sur ces approches de découverte des patterns structurels des données est présentée dans la section 2.4.3.

L'annotation consiste à étiqueter des données afin de restituer leur sens. Dans la section 2.5, nous présentons les approches qui s'intéressent à l'annotation. Ces approches peuvent être classées selon leur contexte en quatre catégories : les approches d'annotation des données liées, qui sont présentées dans la section 2.5.1 ; les approches d'annotation des tables Web qui sont présentées dans la section 2.5.2 ; les approches d'annotation des documents qui sont présentées dans la section 2.5.3 et les approches d'annotation de texte qui sont présentées dans la section 2.5.4. Une discussion sur ces approches d'annotation est présentée dans la section 2.5.5.

2.2 Découverte du schéma implicite

Plusieurs travaux dans la littérature se sont intéressés à la découverte de schéma. Cela, à des fins différentes, comme pour résumer des jeux de données, formuler des requêtes, indexer les données, etc. Certains de ces travaux découvrent

le schéma d'une source de données sans avoir besoin d'aucune information sur le schéma déclarée dans le jeu de données. Ces approches partent d'un jeu de données sans aucune indication sur le schéma, et tentent de caractériser ces données en fournissant un data guide, ou en appliquant un algorithme de clustering pour regrouper les données selon leur similarité.

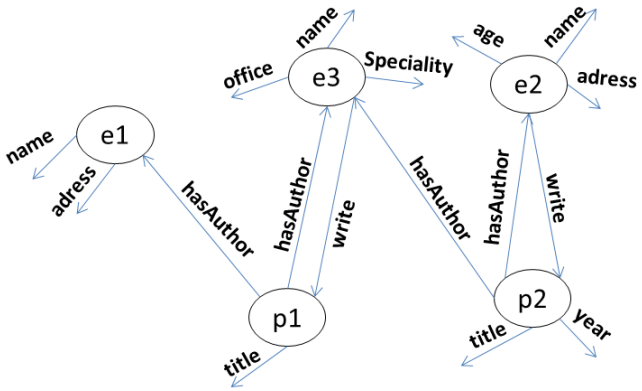


FIGURE 2.2 – Exemple d'un graphe de données

Les approches de découverte de schéma procèdent de deux manières différentes. La première famille d'approches procède par la formation de groupes d'instances qui sont structurellement similaires. La deuxième famille d'approches, regroupe les chemins qui se ressemblent dans le graphe de données.

Les données semi-structurées sont généralement représentées par un graphe, comme par exemple le modèle OEM (Object Exchange Model) [25], qui est un graphe orienté étiqueté, où une donnée peut être atteinte par différents chemins. Il permet de faire face à l'irrégularité et à l'incomplétude des données. Un graphe OEM, comme un graphe RDF, décrit bien les liens entre les objets. En revanche, le manque de sémantique pour un graphe OEM le rend compréhensible par les humains uniquement. Alors qu'un graphe RDF est compréhensible par les machines grâce à un protocole et des langages associés. L'absence de schéma, dans un graphe de données, qu'il soit représenté en OEM ou en RDF, rend son exploitation difficile, comme par exemple pour formuler une requête.

La figure 2.2, représente un graphe de données. Chaque nœud représente une instance comme un étudiant e_1 ou une publication p_2. Chaque arc est orienté et étiqueté par le nom de la propriété qu'il représente. Un arc peut lier deux nœuds représentant deux instances comme l'arc étiqueté *hasAuthor*, ou une instance et un littéral comme l'arc étiqueté *title*. Des objets de même type peuvent être décrits par des ensembles de propriétés différents. Les objets e_1 et e_2 représentent tous les deux des étudiants. Cependant, l'objet e_2 est décrit par une propriété *âge* alors que l'objet e_1 ne l'est pas.

La figure 2.3, illustre le résultat de la première famille d'approches pour

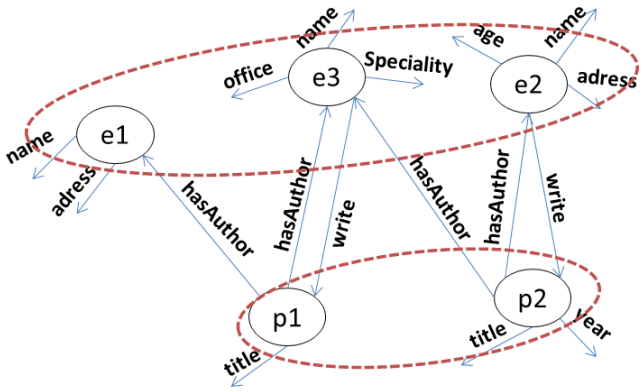


FIGURE 2.3 – Exemple d'un regroupement des instances similaires du jeu de données de la figure 2.2

découvrir le schéma d'un jeu de données. Celle-ci procède par la formation de groupes d'instances qui sont structurellement similaires. Le regroupement des instances de la figure 2.2 donne deux groupes d'instances similaires structurellement. Le premier groupe représente des étudiants et il est formé des instances : e_1, e_2 et e_3. Le deuxième groupe représente des publications et il est formé des instances : p_1 et p_2.

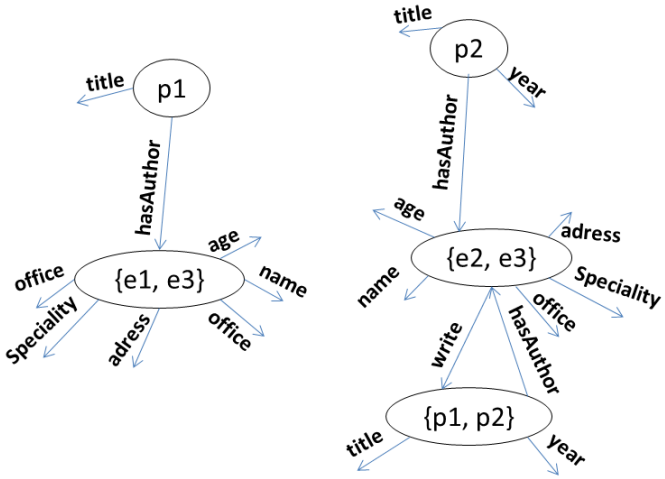


FIGURE 2.4 – Exemple d'un regroupement de chemin du jeu de données de la figure 2.2

La figure 2.4, illustre le résultat de la deuxième famille d'approches qui découvrent le schéma d'un jeu de données. Celles-ci consistent à regrouper les chemins qui atteignent la même donnée. Le regroupement des chemins de la figure 2.2 donne un ensemble de chemins possibles entre les instances. Une instance

peut apparaître plusieurs fois dans le graphe construit, comme l'instance e_3. En revanche, certaines instances ne sont pas regroupées même si elles sont du même type, comme pour les objets e_2 et e_3 qui représentent des étudiants.

Nous présentons dans ce qui suit les approches de découverte de schéma. Les approches qui procèdent par la formation de groupes d'instances qui sont structurellement similaires sont présentées dans la section 2.2.1. Les approches qui regroupent les chemins similaires dans un graphe de données sont présentées dans la section 2.2.2. Une discussion sur ces approches de découverte du schéma implicite d'un jeu de données est présentée dans la section 2.2.3.

2.2.1 Approches de découverte de schéma par regroupement des instances

Les approches décrites dans cette section, découvrent le schéma d'un jeu de données en regroupant les instances similaires structurellement. Celles-ci utilisent différentes techniques comme le clustering ou l'analyse formelle des concepts (FCA), etc.

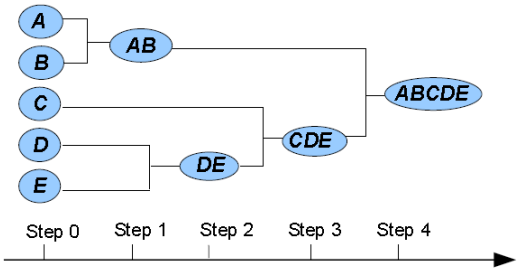


FIGURE 2.5 – Regroupement par l'algorithme hiérarchique ascendant

Approche de Christodoulou, K. et al. (2013) : Inférence de structure [1, 26].

L'approche proposée dans [1, 26] s'appuie sur un algorithme de clustering hiérarchique ascendant standard pour déduire un résumé structurel d'un jeu de données RDF. Le choix de cet algorithme de clustering est dû au fait qu'il n'y a pas de connaissance *a priori* sur le nombre de types contenus dans le jeu de données.

Comme décrit dans la figure 2.6, chaque ressource est représentée par ses propriétés sortantes. Un algorithme de clustering hiérarchique ascendant est ensuite appliqué sur ces ressources pour former des clusters qui représentent des classes/types comme décrit dans la figure 2.5. Une classe est ensuite annotée

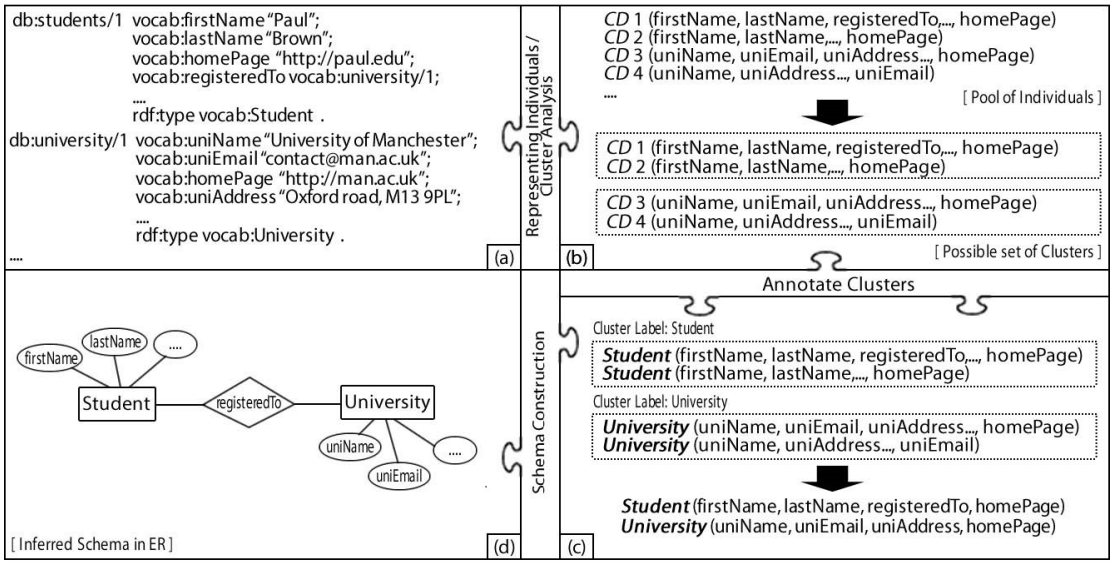


FIGURE 2.6 – (a) Triplets RDF en entrée (b) Représentation des individus et groupement (c) Annotation des clusters (d) Schéma inféré représenté sous la forme d'un diagramme ER [1]

par la valeur la plus fréquente des déclarations *rdf:type* pour les instances d'un groupe. S'il n'y a pas de déclarations *rdf:type* pour les instances de la classe, la classe est dite inconnue. Les attributs d'une classe sont l'union des attributs de tous ses individus, ce qui peut être vu comme un pattern approximatif d'une classe. La similarité de Jaccard, qui reflète le nombre d'attribut en commun, est utilisée pour mesurer la similarité entre deux instances. La similarité entre deux nœuds est calculée comme la moyenne des similarités entre chaque paire d'individus appartenant respectivement à chacun des deux nœuds. Pour déterminer la meilleure partition de l'algorithme hiérarchique, l'approche tente de maximiser la dissimilarité moyenne entre chaque individu et le complément de son cluster, et de minimiser la dissimilarité moyenne entre chaque individu et les individus de son cluster.

Cette approche extrait un schéma à partir de données RDF avec une bonne qualité du schéma découvert. L'algorithme hiérarchique permet effectivement de construire progressivement les types sans connaissances *a priori* sur leur nombre dans le jeu de données. Cependant, l'arbre hiérarchique est parcouru après le clustering pour évaluer le meilleur niveau de coupure. De plus, l'algorithme de clustering hiérarchique est très coûteux en terme de temps, donc peu adapté pour de grands graphes RDF.

Approche de Chen, Jesse Xi et Reformat, Marek (2014) : Découverte de catégories [2].

Les auteurs de [2] proposent d'appliquer un algorithme de clustering hiérarchique sur des données RDF, où à chaque itération les clusters les plus similaires sont regroupés. Une hiérarchie de catégories est alors construite. Le degré d'appartenance de chaque instance à une catégorie donnée est également évalué. La figure 2.7 montre un exemple de clustering hiérarchique.

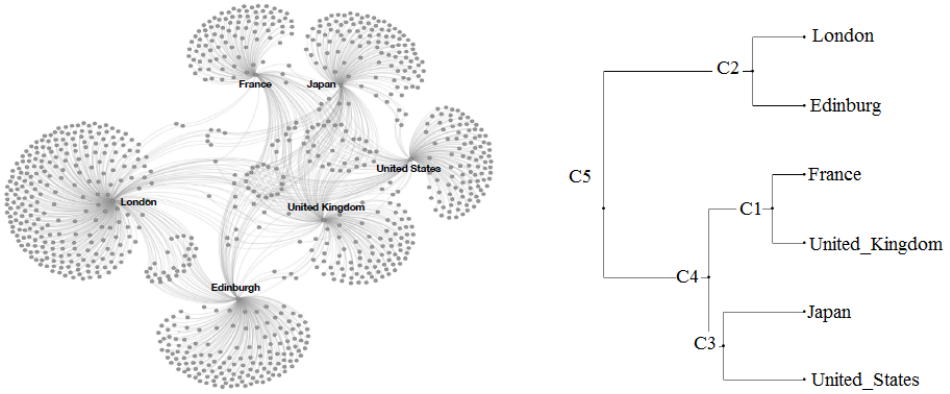


FIGURE 2.7 – Clustering hiérarchique (b) pour la découverte de catégories sur 6 instances (a) [2]

L'approche construit une matrice de similarité sur les données. La similarité entre deux instances est calculée à l'aide d'une mesure de similarité inspirée de l'indice de Jaccard [27], elle reflète le nombre d'instances comme valeurs objets des propriétés partagées entre les deux instances, c'est à dire, la proportion des propriétés des deux instances qui sont équivalentes et qui ont les même valeurs de ressources lorsque ces dernières ne sont pas des littéraux. Elle est évaluée comme le nombre de propriétés identiques ayant les mêmes objets sur le nombre de l'union des propriétés des deux instances.

Les valeurs de similarité entre les paires d'instances de la même catégorie sont différentes. Cela implique qu'une instance RDF appartient à sa catégorie dans une certaine mesure. Pour évaluer le degré d'appartenance d'une instance à sa catégorie, le medoïd de chaque cluster est d'abord identifié. Le medoïd est l'objet dont la similarité avec les autres objets du cluster est la plus grande. Le degré d'appartenance d'une instance à une catégorie est évalué comme le rapport entre la similarité de l'instance avec les autres instances du cluster, et la similarité du médoïd avec les autres instances du cluster.

Cette approche introduit une méthodologie qui permet de découvrir les catégories de données RDF. Un clustering hiérarchique est utilisé pour cela, ainsi qu'une évaluation du degré d'appartenance d'une instance à une catégorie. La

catégorie d'une instance ne reflète pas forcément le type de l'instance. En raison de la mesure de similarité utilisée, la catégorie est plus restrictive qu'un type. En effet, deux instances de même type n'ont pas à avoir un ensemble similaire d'objets (valeurs) pour leurs propriétés. L'approche repose sur la construction d'une grande matrice de similarité entre les instances pour les différentes étapes de l'approche, telles que le clustering et l'évaluation du degré d'appartenance d'une instance, ce qui n'est pas adapté pour un large jeu de données.

Approche de M. Kirchberg et al. (2012) : Découverte de concepts formels [28].

L'approche présentée dans [28], utilise l'analyse formelle des concepts (FCA) [29] pour trouver les concepts à partir des données du Web sémantique. FCA permet de structurer les données sous forme d'une représentation en treillis afin de les analyser.

La figure 2.8 représente une table décrivant un jeu de données G = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, et l'ensemble des propriétés M = {*composite*, *even*, *odd*, *prime*, *square*}.

	composite	even	odd	prime	square
1			✓		✓
2		✓		✓	
3			✓	✓	
4	✓	✓			✓
5			✓	✓	
6	✓	✓			
7			✓	✓	
8	✓	✓			
9	✓		✓		✓
10	✓	✓			

FIGURE 2.8 – Exemple d'une table décrivant un jeu de données

La figure 2.9 représente le treillis de concepts issu de l'analyse FCA pour les données de la figure 2.8. L'ensemble complet de concepts pour les objets et les attributs est représenté dans ce treillis. Il comprend un concept pour chacun des attributs *composite*, *even*, *odd*, *prime* et *square*. De plus, il comprend des concepts pour des nombres composés pairs (c,e), des nombres composés carrés (c,s), des carrés impairs (o,s), des carrés composés pairs (c, e, s), des carrés composés impairs (c, o, s), des nombres premiers pairs (e, p) et des nombres premiers impairs (o, p).

L'analyse et la visualisation des données dans un treillis peut se faire par objet et par attribut : le plus petit concept incluant le numéro 3 est celui avec les objets

{3, 5, 7} et les attributs {*odd*, *prime*} ; le plus grand concept impliquant l'attribut *square* est celui avec des objets {1,4,9}, car 1, 4 et 9 sont tous des nombres carrés.

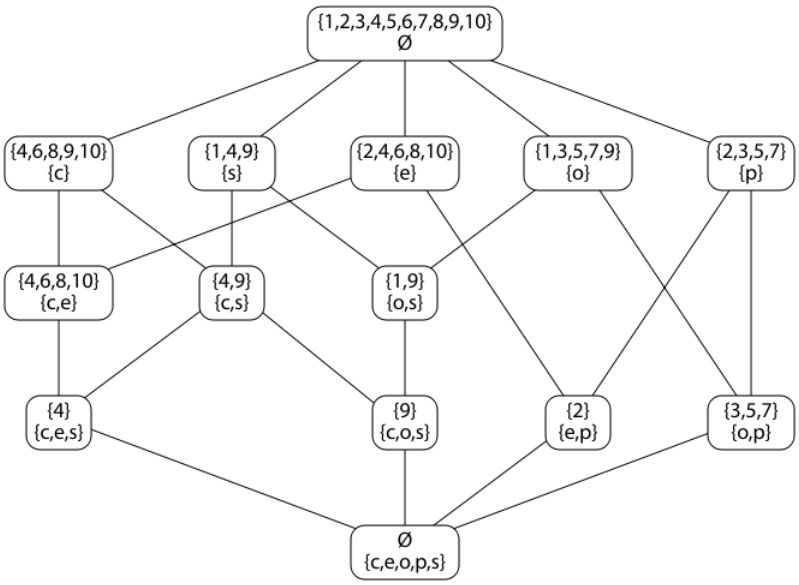


FIGURE 2.9 – Treillis de concepts pour les instances de la figure 2.8, et leur propriétés : composite (c), square (s), even (e), odd (o) et prime (p)

Cette approche regroupe des instances qui partagent un certain nombre de propriétés. Le schéma du jeu de données est représenté dans un treillis. Ceci est une modélisation intéressante d'un schéma qui permet d'analyser et de visualiser les données par instance et par attribut. Cependant, une instance peut apparaître plusieurs fois dans le treillis. Chaque nœud dans un treillis ne représente pas forcément un type. Il reflète juste le fait qu'un ensemble de propriété données décrit un ensemble d'instances données. Les liens entre les nœuds d'un treillis ne reflètent pas les liens entre des types. De plus, les diagrammes en treillis peuvent être de très grande taille car ils peuvent contenir jusqu'à 2^m x 2^n concepts potentiels, où m est le nombre d'attributs et n le nombre d'objets. Un treillis peut être vu plus comme un index de schéma pour filtrer et mapper des requêtes que comme un schéma décrivant les types et liens entre eux pour aider à formuler des requêtes.

Approche de Brosius, Dominik et Staab, Steffen (2016) : Découverte de schéma sous la forme d'un index avec FCA [30].

L'idée des auteurs dans [30] est de définir le schéma d'une source de données RDF comme une structure d'index sur les données. Cet index est utilisé

lors de l'interrogation des données. L'analyse formelle des concepts (FCA) est utilisée pour la construction de l'index, car cette méthode est indépendante des facteurs externes, tels que l'implication des experts ou la spécification de paramètres comme pour le clustering. Le treillis de concepts issu de l'analyse FCA des données va représenter l'index de schéma.

Cet index de schéma peut être utilisé comme un premier filtre sur les sous-graphes de données qui peuvent potentiellement répondre à une requête. Cependant, des sous-graphes non pertinents peuvent être renvoyés car l'index ne contient pas d'information sur les valeurs des propriétés. Par conséquent, l'index de schéma proposé n'est pas nécessairement optimal. En perspectives, les auteurs prévoient de réduire le nombre de sous-graphes non pertinents qui sont retournés pour une requête, cependant, les optimisations potentielles peuvent être au détriment du coût de construction de l'index et de la recherche dans l'index.

En ce qui concerne la réalisation de l'index de schéma proposé, les auteurs s'attendent à des difficultés sur la taille du treillis et de son évolution de manière incrémentale.

Approche de S. Nestorov et al. (1997) : Inférence de structure pour les données semi-structurées [3].

L'approche proposée dans [3] permet de découvrir la structure des données semi-structurées au format OEM [25]. L'approche traite le problème de l'identification de certaines structures sous-adjacentes dans de grandes collections de données semi-structurées. Comme les données sont assez irrégulières, cette structure se compose d'une classification approximative des objets en une collection hiérarchique de types. La méthode introduit la notion de hiérarchie de type pour ces données, ainsi qu'un algorithme pour dériver la hiérarchie des types, et des règles pour attribuer des types aux données.

Pour appliquer cette approche, tous les arcs entrants d'un objet o sont désignés par $role(o)$ et tous leurs arcs sortants sont désignés par $attribute(o)$. Pour un ensemble d'arcs s, $at(s)$ représente le nombre d'objets o tels que : $attribute(o) = s$. Le nombre d'objets o tel que $s \subseteq attribut(o)$, est désigné par $above(s)$. Le concept de *saut* permet d'évaluer l'homogénéité que peut avoir un ensemble d'objets avec un ensemble d'arcs s comme suit :

$$Sauter(s) = \frac{at(s)}{above(s)}$$

L'approche proposée est composée des étapes suivantes :

1. Identification des types candidats : le graphe L est construit (voir figure 2.11) à partir du jeu de données D (voir figure 2.10), de sorte qu'un nœud représente l'ensemble de $attribut(o)$ et le nombre de o qui ont ces attributs. Un arc est une inclusion entre deux ensembles. Les nœuds dont le *saut* est

au-dessus d'un seuil fixé par l'utilisateur sont considérés comme des types candidats. Plus de types candidats sont retrouvés en regroupant les nœuds qui ne représentent pas des types candidats et en tolérant une certaine hétérogénéité des ensemble des arcs des objets.

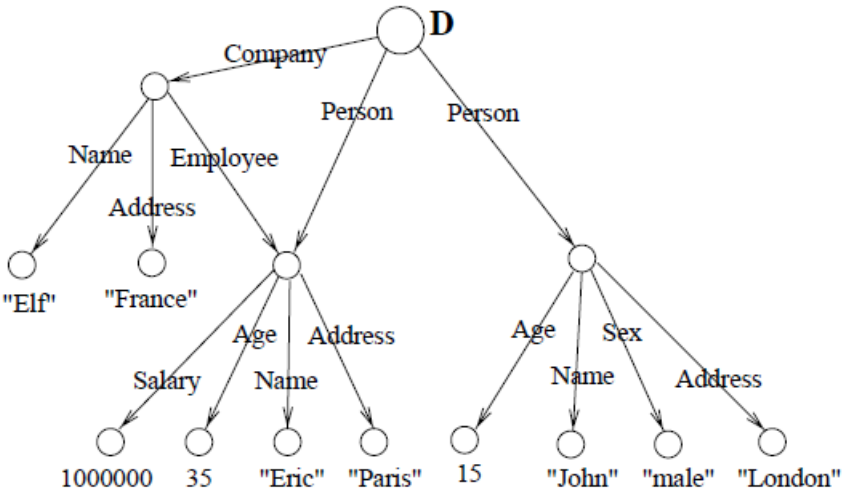


FIGURE 2.10 – Exemple d'un graphe OEM [3].

2. Sélection des types et des sous-types des candidats et leur organisation dans une hiérarchie de type : un type candidat S est marqué par l'arc entrant le plus fréquent ($roles(o)$) des objets o qui appartiennent à S, noté rôle principal : $p-role(S)$. Un candidat T est considéré comme un type s'il n'y a pas d'autres candidats T' tel que $T' \subset T$. Un candidat S est considéré comme un sous-type s'il n'est pas un type et il n'y a pas d'autre candidat S' tel que $S' \subset S$ et $role-p(S) = p-role(S')$.
3. Déduction des règles de typage : un objet est de type T s'il a les mêmes attributs que lui. Si aucun type n'a les mêmes attributs que l'objet, alors considérer tous les types et sous-types qui ont le même $p-role$ que l'objet et calculer la distance entre l'objet et le type candidat, tel que la distance soit égale au "nombre d'attributs différents". Un objet peut avoir plusieurs types par rapport à la hiérarchie.
4. Vérification de la hiérarchie de type par rapport aux données en évaluant la précision.

La méthode déduit implicitement les structures et les types de données semi-structurées de façon incrémentale et déterministe. Elle permet de découvrir plusieurs types pour un objet, mais uniquement au sens de la hiérarchie de généralisation, c'est-à-dire, non pas dans le sens où un objet est un *employé* et un *joueur*, mais qu'un objet est un *employé* et une *personne*. L'approche distingue

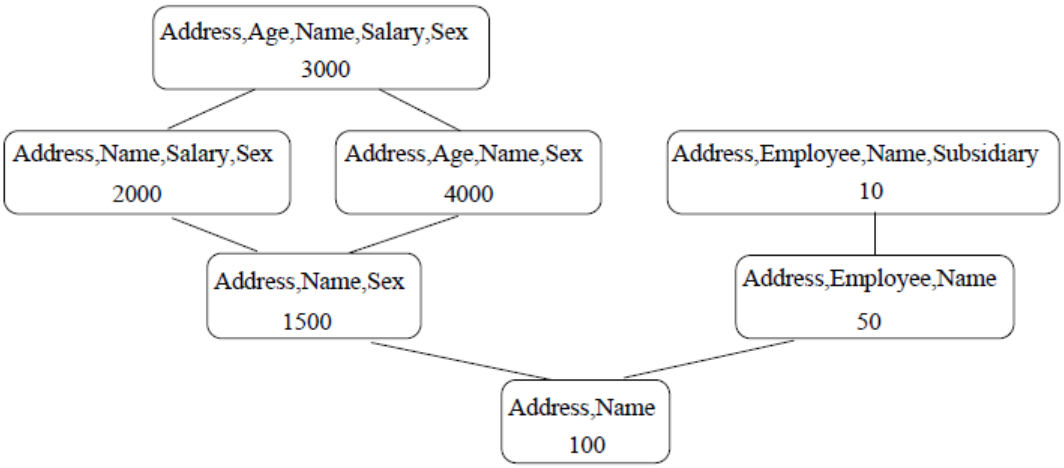


FIGURE 2.11 – Le comptage treillis L construit à partir du jeu de données D de la figure 2.10 [3].

les arcs entrants et sortants : les arcs entrants sont considérés comme des rôles (étiquettes potentielles pour le type). Ce qui est applicable au modèle OEM contrairement à RDF, où les arcs entrants ne reflètent pas nécessairement le type d'une instance. En outre, les données semi-structurées représentées par le modèle OEM n'ont pas autant de sémantique que des données RDF par la signification de certaines propriétés telles que : label, sameAs, etc. Par conséquent, toutes les propriétés dans une source de données RDF ne doivent pas être traitées de la même manière. L'algorithme proposé nécessite de fixer le seuil du saut (comparable à un seuil de similarité) qui n'est pas facile à définir car il dépend de la régularité des données. Pour une valeur de saut à 1, la construction du graphe L revient à regrouper des objets avec un seuil de similarité à 1. En effet, chaque structure d'objet est comparé à la structure d'un nœud dans le schéma, et si aucun nœud ne correspond, un nouveau nouveau noeud est créé pour représenter la structure de l'objet. Cela est coûteux et ne convient pas pour une grande masse de données.

Approche de S. Nestorov et al. (1998) : Extraction de schéma à partir de données semi-structurées [31].

Les auteurs de l'approche [3] proposent de l'améliorer notamment car elle n'est pas robuste au bruit et elle n'est pas adaptée pour les sources de données volumineuses. Ils suggèrent dans [31] d'appliquer l'algorithme de clustering k-means pour l'extraction de schéma à partir de données semi-structurées. Cependant, l'utilisation de l'algorithme de clustering k-means requiert le nombre de clusters voulus, ce qui n'est pas évident à définir, car on ne connaît pas *a priori* le nombre de types contenus dans le jeu de données. La méthode forme effectivement des types par regroupement, mais elle n'est pas incrémentale. L'approche fait une

distinction entre les arcs entrants et sortants : les arcs entrants sont considérés comme des rôles et des étiquettes potentielles pour les types inférés, ce n'est pas le cas pour des données RDF où les arcs entrants ne reflètent pas nécessairement le type d'une instance.

2.2.2 Approches de découverte de schéma par regroupement des chemins

Les approches présentées dans la section précédente regroupent les instances selon leur similarité structurelle. Une autre façon de faire émerger le schéma implicite d'un graphe de données est de regrouper ses chemins similaires. Nous présentons dans cette section les approches de découverte de schéma implicite par regroupement des chemins. Celles-ci visent à identifier des patterns de chemins qui caractérisent un graphe de données.

Approche de Goldman R, et al. (1997) : Dataguides [32].

Le dataguide [32] est une représentation arborescente de la structure du jeu de données, dans laquelle, chaque chemin du graphe initial apparaît exactement une seule fois. Il a été proposé essentiellement pour interroger les données semi-structurées, mais il sert également à résumer le jeu de données.

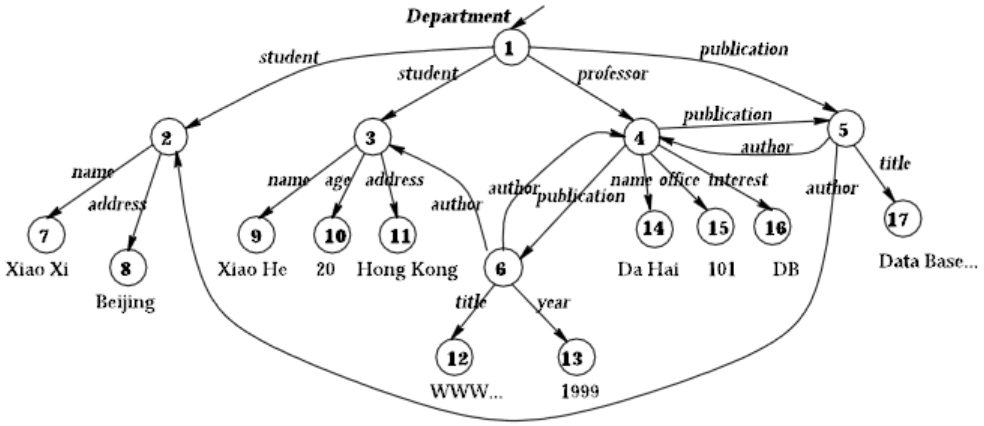


FIGURE 2.12 – Un graphe OEM [4]

La figure 2.12 montre un graphe de données OEM (Object Exchange Model) [25]. Les objets 2 et 3 représentent tous les deux des étudiants. Cependant, l'objet 3 est décrit par une propriété *âge* alors que l'objet 2 ne l'est pas. Comme autre exemple, nous avons les objets 5 et 6 qui représentent tous les deux des publications ; et pourtant, ces deux objets de même type sont à des niveaux différents dans le graphe de données OEM.

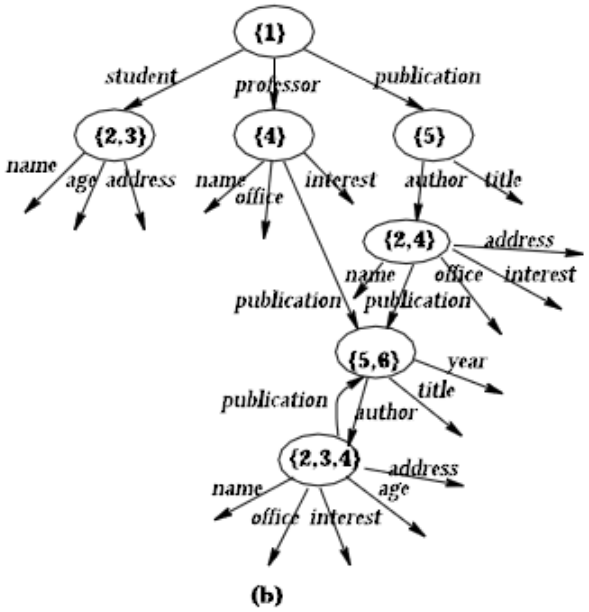


FIGURE 2.13 – Le dataguide du graphe de données de la figure 2.12 [4]

Le dataguide d'un jeu de données est construit en fusionnant les chemins qui atteignent la même donnée, comme le montre la figure 2.13. Ce qui permet de regrouper certaines instances similaires. Des propriétés peuvent être affectées à un nœud même si toutes ses instances n'ont pas cette propriété, comme pour la propriété *âge* qui est affectée au nœud contenant les objets 2 et 3, alors que l'objet 2 n'est pas décrit par cette propriété. Des nœuds de même type peuvent apparaître plusieurs fois dans un dataguide, par exemple les nœuds {5} et {5, 6} de type *Publication*.

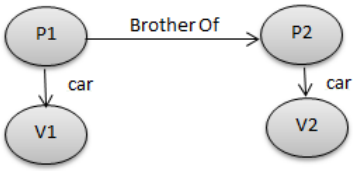


FIGURE 2.14 – Exemple d'objets du même type non regroupés dans un dataguide

Dans un dataguide certaines instances ne sont pas regroupées même si elles sont du même type, comme pour les objets $P1$ et $P2$ qui représentent des personnes dans la figure 2.14. En effet, un dataguide représente plus un plan d'accès aux données que des patterns de types, mais malgré cela, un dataguide

permet de donner des informations sur le contenu d'un jeu de données.

Approche de Wang, Q. et al. (2000) : Dataguide approximatif [4].

Un DataGuide est précis mais sa taille peut être beaucoup plus grande que le graphe de données initial. L'approche présentée dans [4], propose un dataguide approximatif fondé sur une méthode de clustering hiérarchique incrémentale (COBWEB [33]) qui regroupe les sommets ayants des arcs entrants et sortants similaires. Au départ, le schéma approximatif, noté Gs, ne contient que la racine du graphe de données, noté Gd, puis Gd est parcouru pour ajouter ses nœuds dans Gs soit à un nœud existant, soit en créant un nouveau nœud selon la valeur de la fonction d'utilité du clustering. Un arc est ajouté entre les deux nœuds s'il n'existait pas dans Gs. Ce schéma approximatif, représenté dans la figure 2.15, est moins grand qu'un dataGuide ce qui optimise le temps de traitement des requêtes. Cependant, il peut contenir des chemins en double ou qui n'existent pas dans le graphe des données.

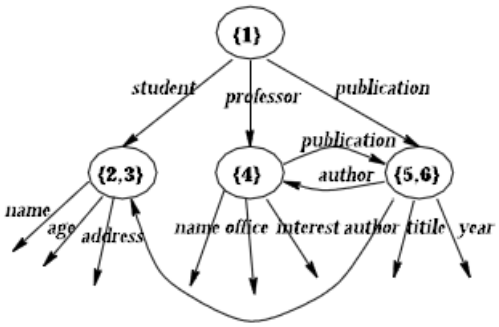


FIGURE 2.15 – Le schéma approximatif du graphe de données de la figure 2.12 [4]

La démarche proposée dans [4] pour extraire le schéma du graphe de données, est applicable pour construire des types pour un graphe RDF. Cela en considérant un nœud du graphe de schéma Gs comme un type candidat qui contient ses instances. Cependant, la méthode ne fait pas de distinction entre un arc entrant ou sortant d'un objet lors du regroupement. Elle considère les deux d'arcs de la même façon comme une propriété d'un objet, ce qui peut fausser le résultat du regroupement comme le montre la figure 2.16, où l'objet $author_1$ de type $Author$ sera regroupé avec l'objet $presentaion_1$ de type $Presentation$, car ils ont les mêmes arcs entrants/sortants. Ceci rend l'approche moins adaptée au cas de données RDF où il est important de différencier le domaine du co-domaine pour former des types. La méthode n'est pas très sélective pour la construction de types, car il y a un compromis entre la

taille du graphe (nombre minimum de types obtenus) et la précision (nombre maximum de types obtenus) dans la fonction d'utilité lors du regroupement, alors que pour la découverte du schéma, la taille du graphe n'est pas un problème, le plus important est la précision de la qualité des types découverts. L'algorithme COBWEB utilisé pour le regroupement n'est pas déterministe, le résultat change selon le parcours du graphe de données pour construire le schéma.

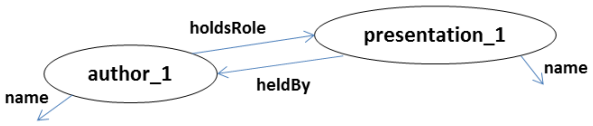


FIGURE 2.16 – Exemple de regroupement de deux objets de types différents

Approche de Schätzle, A. et al. (2013) : Bisimulation sur un graphe RDF [34].

L'approche proposée dans [34] vise à rendre un graphe RDF plus compréhensible en réduisant sa taille. L'idée de cette approche est de découvrir des sous-graphes qui sont similaires en considérant leur structure. Elle utilise des techniques de bisimulation [35] pour regrouper les chemins de sous-graphes similaires.

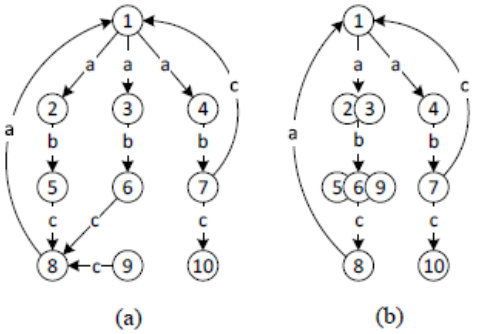


FIGURE 2.17 – Graphe RDF simplifié (a) et sa bisimulation indiquée par une représentation réduite (b).

Intuitivement, deux nœuds dans un graphe sont bisimilaires s'ils ont les mêmes arcs sortants et leurs successeurs également. Les arcs sortants des nœuds bisimilaires sont regroupés. La figure 2.17 montre un graphe RDF (a) et sa bisimulation indiquée par une représentation réduite (b). Les nœuds 2 et 3 sont bisimilaires car ils ont le même arc sortant *b*, et leurs successeurs les nœuds 5 et 6 ont également le même arc sortant *c*. Cependant, le nœud 4 n'est pas bisimilaire avec les nœuds 2 et 3, et ce même s'il a le même arc sortant *b*, car son fils le nœud 7 n'est pas

bisimilaire avec les nœuds 5 et 6 car il a deux arcs sortants c alors que ces derniers n'en ont qu'un seul.

Les nœuds générés contiennent des objets qui ont exactement les mêmes propriétés sortantes, alors que dans un graphe RDF, des instances de même type peuvent être décrites par des ensembles de propriétés différents.

Approche de Delteil, A. et al. (2001) : Découverte d'ontologies à partir des déclarations RDF [5].

La généralisation de chemin à partir des déclarations RDF est utile pour organiser les connaissances et découvrir l'ontologie. Delteil et al. proposent dans [5] une méthode incrémentale qui permet d'obtenir une généralisation de chemin à partir de déclarations RDF.

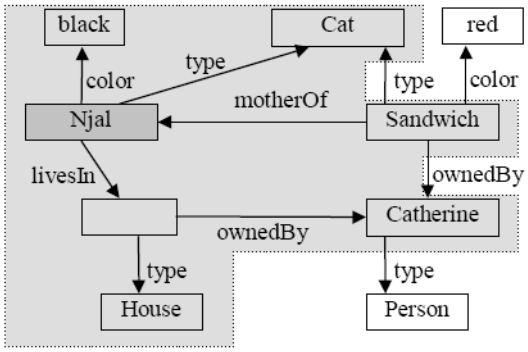


FIGURE 2.18 – Un extrait du graphe RDF décrivant la ressource "Njal" [5]

La longueur des descriptions de ressources partielles est augmentée progressivement. Une hiérarchie de généralisation S_1 est d'abord construite à partir des descriptions de ressources de longueur 1. S_n est ensuite construit à partir de S_{n-1} et S_1 en augmentant progressivement la longueur maximale des descriptions de ressources.

Au départ, l'ensemble de description de taille 1 annoté $D1(R)$ pour une ressource R est construit. $D1(R)$ est l'ensemble des triplets RDF dont l'objet ou le sujet est R. La figure 2.18 représente une partie d'un graphe RDF concernant la ressource "Njal". La partie grisée de ce graphe représente l'ensemble de description $D1(Njal)$. La longueur des descriptions de ressources est augmentée progressivement. Une hiérarchie de généralisation S_1 est d'abord construite à partir des descriptions de ressources de longueur 1. Une hiérarchie de généralisation S_n de taille n est ensuite construite à partir de S_{n-1} et S_1 en augmentant progressivement la longueur maximale des descriptions de ressources. Le principe de construction de S_1 et le principe de construction de S_n à partir de S_{n-1} et S_1 sont détaillés ci-après.

Le principe de construction de S_1 est le suivant :

1. L'extraction de l'ensemble de description $D1(R)$ de longueur 1 pour toute ressource R, à partir du graphe RDF. $D1(R)$ est l'ensemble des triplets RDF dont l'objet ou le sujet est R.
2. La généralisation itérative de toutes les paires possibles de triplets pour construire $L1$, l'ensemble des triplets généralisés. La généralisation de deux triplets $(R1, P1, V1)$ et $(R2, P2, V2)$ est un triplet plus général (RG, PG, VG) tel que si PG est une propriété généralisant $P1$ et $P2$, et si $V1$ et $V2$ sont des types, alors VG est leur super-type, par exemple :
   - (Catherine, type, person), (Njal, type, cat) $\overset{généralisation}{\rightarrow}$ ({catherine, Njal}, type, living being) ; Notons que "living being" est déduite manuellement.
3. La généralisation peut générer des triplets partageant un même sujet ou objet. Ces triplets sont regroupés en un seul, par exemple :
   - (Njal, color, Black) $\overset{généralisation}{\rightarrow}$ (Njal, color, *)
   - (Sandwich, color, red) $\overset{généralisation}{\rightarrow}$ (Njal, color, *)
   - Regroupement des deux triplets précédents comme suit : ({Njal, Sandwich}, color, *)
4. L'élimination des doublons après regroupement : ce qui consiste à construire S_1 sur la base des relations d'inclusion entre les sujets de nœuds. Plusieurs nœuds peuvent partager le même objet. Dans ce cas, le nœud correspondant au sujet le plus général est conservé, par exemple :
   - ({Njal, Catherine}, type, Living Being)
   - ({Njal, Catherine, Sandwich}, type, Living Being)

   Le premier triplet est éliminé car il est inclus dans le deuxième.

La figure 2.19 montre la généralisation et le regroupement des chemins de taille 1 pour la figure 2.18. Nous pouvons voir dans cette figure les chemins communs entre différentes ressources, par exemple : les ressources *Njal* et *Sandwich* ont en commun les chemins : ({Njal, Sandwich}, type, Cat) et ({Njal, Sandwich}, color, *).

Le principe de construction de S_n à partir de S_{n-1} et S_1 est le suivant :

1. La construction itérative de l'ensemble des triplets généralisés L_n de taille n, par jointure de toutes les paires possibles d'un triplet généralisé L_1 de taille 1 et un triplet de chemin de longueur $n-1$ de L_{n-1}. Deux triplets peuvent être joints si l'objet du premier triplet est égal au sujet du deuxième triplet. Un chemin de longueur n est le résultat de $n-1$ jointures entre n triplets.
2. Le regroupement triplets partageant un même sujet ou objet : la généralisation peut générer des triplets partageant un même sujet ou objet. Ces triplets sont regroupés en un seul. Cette étape est analogue à l'étape 3 de la construction de S_1.

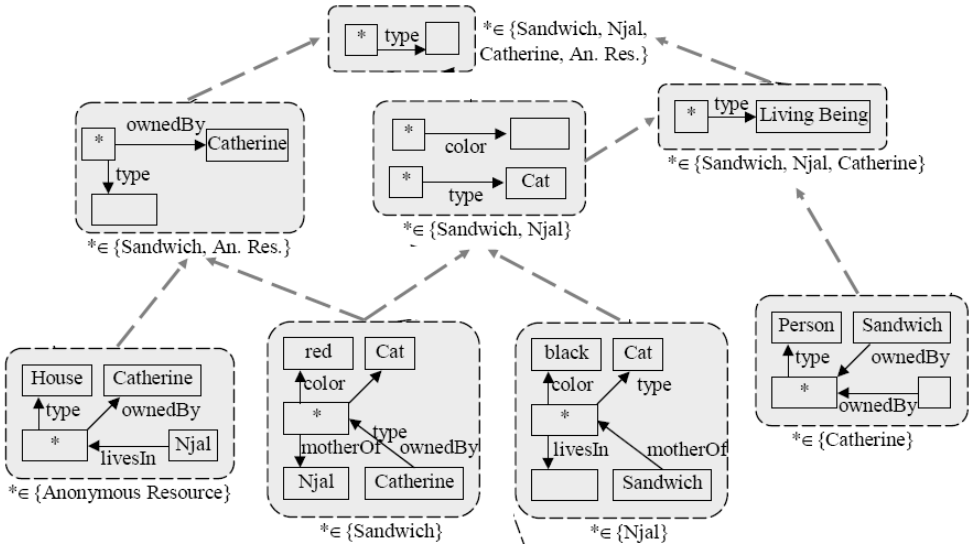


FIGURE 2.19 – Généralisation et regroupement des chemins de taille 1 de la figure 2.18 [5]

3. L'élimination des doublons après regroupement : construction de S_n sur la base des relations d'inclusion entre les nœuds sujets. Cette étape est analogue à l'étape 4 de la construction de S_1.

L'approche fait un résumé des déclarations RDF d'un jeu de données en les généralisant, et ce en regroupant les chemins communs. Afin de faire face à la complexité d'une telle tâche, la hiérarchie est construite progressivement en augmentant à chaque étape la taille (le chemin) des descriptions de ressources RDF. Le but de la méthode est de généraliser les chemins d'un ensemble de déclarations RDF afin de les regrouper. Cependant, la méthode ne propose pas d'approche automatique de généralisation, par exemple, la généralisation de « cat, person » en « living being » est faite manuellement.

2.2.3 Discussion sur les approches de découverte du schéma implicite

Les approches qui découvrent le schéma d'un jeu de données à partir de la structure implicite des données ne font aucune hypothèse sur l'existence de déclarations sur le schéma du jeu de données. Certaines de ces approches [1, 26, 2, 28, 30, 3, 31] procèdent par le regroupement des instances similaires, tandis que d'autres [32, 4, 5, 34] procèdent par le regroupement des chemins similaires dans le graphe de données. Ces approches sont synthétisées respectivement dans les tableaux 2.1 et 2.2.

Les approches [1, 26, 2, 28, 30, 3, 31] qui regroupent les instances selon leur similarité structurelle pour découvrir des types présentent la limite majeure liée

TABLE 2.1 – Synthèse des approches de découverte du schéma implicite par regroupement des instances

Approches	Description d'une entité	Paramètres	Similarité	Incrémental	Typage multiple	Label pour les clusters	Résultats
Christodoulou et al [1,26] (clustering hiérachique)	Propriétés sortantes	Seuil de coupure dans l'arbre hiérarchique (peut être trouvé en parcourant l'arbre mais cela est très couteux)	Jaccard	Non	Non	Déclaration "rdf:type" la plus fréquente dans un cluster	• Types • Liens hiérarchiques et sémantiques entre les types
Chen et al. [2] (clustering hiérachique)	Propriétés sortantes		Jaccard avec objets identiques des propriétés	Version incrémentale proposée	Non	Déclarations "rdf:type" et "dcterm:subject" identiques dans un cluster	• Catégories • Liens hiérarchiques
Kirchberg et al. [28] (FCA)	Propriétés sortantes		Équivalent à identité	Non	Une entité peut appartenir à plusieurs concepts, Cependant, un concept ne reflète pas forcément un type	Non	• Treillis de concepts
Nestorov et al. [3]	Propriétés sortantes	Seuil du saut	Formule proposée	Non	Non	Etiquette de l'arc entrant le plus fréquent	• Treillis de types avec des liens de hiérarchie
Nestorov et al. [31] (regroupement par clustering, ex: k-means)	Propriétés entrantes et sortantes	Nombre de clusters si k-means appliqué	Formules proposées		Non	Etiquette de l'arc entrant le plus fréquent	• Types

aux paramètres à spécifier ; à l'exception des approches qui utilisent FCA [28, 30] et de l'approche [2] qui construit un arbre hiérarchique et qui ne spécifie pas son paramètre de coupure. Dans toutes ces approches, une ressource est représentée par ses propriétés sortantes uniquement, le co-domaine des propriétés n'est donc pas pris en considération pour former les types. Dans l'approche présentée dans [31], les arcs entrants sont considérés uniquement pour trouver un label aux clusters, ce qui équivaut à considérer les déclarations *rdf*:*type* [1, 26, 2] ou *dcterm*:*subject* [2] les plus fréquentes dans un cluster. Les approches proposées ne sont pas incrémentales, sauf [2] pour l'algorithme de clustering hiérarchique pour la découverte des catégories. Les approches proposées ne considèrent pas non plus le fait qu'une instance peut être de plusieurs types, et donc elle peut appartenir à plusieurs groupes. Différents résultats sont fournis par ces approches : des types et des liens sémantiques et hiérarchiques entre eux [1, 26], des catégories et des liens hiérarchiques entre elles [2], un treillis de concepts [28, 30], un treillis de types avec des liens de hiérarchie [3] et des types [31].

Certaines approches tentent de regrouper les chemins du graphe de données dans le but de construire un dataguide [32] ou un dataguide approximatif [4]. Le problème majeur du dataguide [32] est sa taille, car il doit représenter tous les chemins possibles contenus dans le graphe de données. Le dataguide approximatif [4] permet effectivement de réduire la taille du dataguide, cependant, l'approche

	Démarche	Résultats	Taille résultat	Remarques
Goldman et al [32] DataGuide	• Fusion des chemins qui atteignent la même donnée • Chaque chemin n'apparait qu'une fois	Un plan des chemins	Peut-être plus grand que le graphe de données	• Certaines instances de même type ne sont pas regroupées • Une instance peut apparaitre dans plusieurs nœuds
Wang, Q. et al. [4] Approximate Dataguide	• COBWEB • Pas de distinction entre un arc sortant et entrant pour une entité • Une entité n'appartient qu'à un seul nœud	Un plan des chemins	Moins grand qu'un dataguide	• Instances de types différents regroupées • Non déterministe
Schatzle, A. et al. [34]	• Bisimulation : fusion des nœuds ayant les mêmes arcs sortants et leurs enfants également	Un plan des chemins	Moins grand que le graphe des données	• Certaines instances de même type ne sont pas regroupées
Delteil, A. et al. [5]	• Construction itérative de chemins généralisés	Une hiérarchie de chemins généralisés	Très grande taille	• Généralisation semi-automatique

TABLE 2.2 – Synthèse des approches de découverte du schéma implicite par regroupement des chemins

proposée est basées sur l'algorithme COBWEB qui n'est pas déterministe et ne permet pas d'affecter plusieurs types à une instance. L'approche proposée dans [34] utilise la bisimulation pour réduire la taille d'un graphe de données. Cependant, les nœuds générés contiennent des objets qui ont exactement les mêmes propriétés sortantes, alors qu'en fait dans un graphe RDF, des instances de même type peuvent être décrites par des ensembles de propriétés différents. La méthode proposée dans [5] fait un résumé des déclarations RDF d'un jeu de données en les généralisant. Cependant, la méthode ne propose pas une approche automatique de généralisation.

Nous constatons que les approches qui découvrent le schéma implicite d'une source de données en regroupant les instances similaires structurellement sont plus adaptées au contexte de la découverte du schéma d'un graphe de données comme RDF en terme de types et de liens entre eux. En effet, les approches qui regroupent les chemins en commun sont plus adaptées aux données présentées sous forme arborescente en fournissant un plan d'accès.

2.3 Enrichissement du schéma explicite

Dans la section précédente, nous avons vu un ensemble d'approches dont le but est de découvrir le schéma implicite des données sans faire aucune hypothèse sur les déclarations explicites sur le schéma fournies dans le jeu de données. Dans cette section, nous présentons des approches qui utilisent les déclarations sur le schéma pour les compléter ou les enrichir. Nous désignons ces approches par les approches d'enrichissement du schéma existant.

La figure 2.2, représente un graphe de données avec certaines déclarations sur le schéma comme : rdf:*type*, $rdfs$:*range*, $rdfs$:*domain*, etc. Cependant, ces

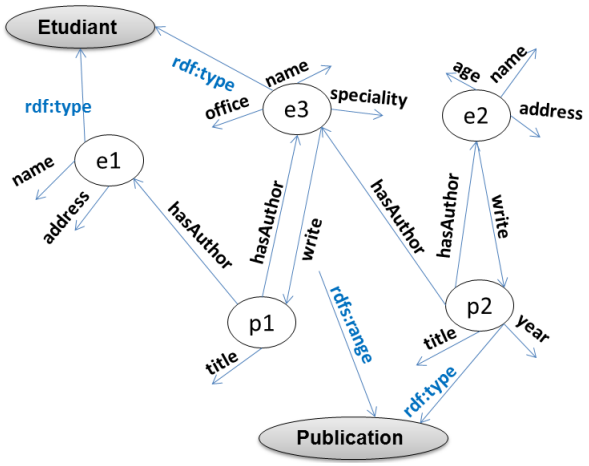


FIGURE 2.20 – Exemple d'un graphe de données avec des déclarations partielles sur le schéma

déclarations peuvent être incomplètes comme la déclaration sur le type pour l'instance p_1.

La figure 2.21, illustre l'enrichissement d'un schéma existant avec des déclarations complémentaires sur le schéma. L'instance p_1 n'a pas de déclaration de type dans le jeu de données, les approches d'enrichissement d'un schéma existant vont tenter d'inférer la déclaration de type de cette instance en se basant sur les déclarations sur le schéma fournies. Par exemple, l'instance p_1 a une propriété entrante *write*, et dans le schéma fourni, cette propriété a une déclaration *rdfs:range* sur le type *Publication*. Ce qui induit que l'instance p_1 peut être de type *Publication*.

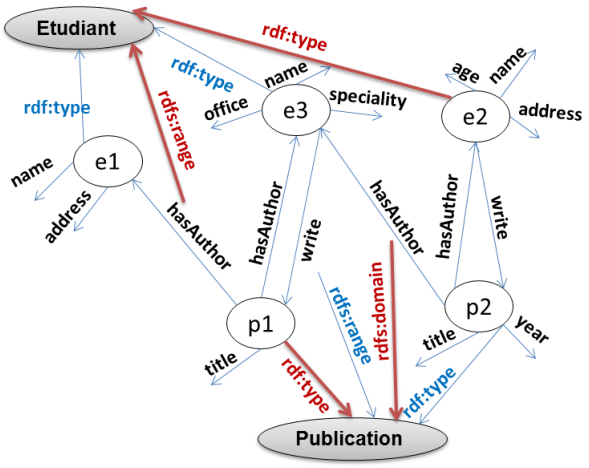


FIGURE 2.21 – Inférence de déclarations complémentaires sur le schéma.

Nous présentons dans ce qui suit les approches qui enrichissent le schéma existant en inférant des déclarations complémentaires sur le schéma d'un jeu de

données. Ces approches peuvent être classées en deux catégories : (i) les approches qui procèdent par fouille de données, présentées dans la section 2.3.1 et (ii) les approches qui font de l'analyse statistique sur les données, présentées dans la section 2.3.2. Une discussion sur ces approches d'enrichissement d'un schéma existant est présentée dans la section 2.3.3.

2.3.1 Approches d'enrichissement par fouille de données

Les approches présentées dans cette section utilisent des méthodes de fouille de données sur les déclarations explicites sur le schéma dans un jeu de données afin de les compléter et de les enrichir.

Approche de Volker, J. et al. (2011) : Induction de schéma [36].

Völker, et al. [36] utilisent l'induction statistique de schéma sur des sources de données RDF, afin de les enrichir. Des règles d'association sont inférées sur les triplets RDF pour acquérir des connaissances au niveau du schéma. L'extraction des règles d'association [37] à partir d'une source de données est une méthode non supervisée de data mining. Elle vise à trouver des corrélations entre les attributs. Les règles d'association qui satisfont un seuil de confiance fourni par l'utilisateur contribuent à la construction du schéma.

	C_1	C_2	C_3	P_{d1}	P_{d2}	P_{d3}	P_{r1}	P_{r2}	P_{r3}
e_1	0	1	0	1	0	1	1	1	0
e_2	1	0	1	0	1	1	1	0	1
e_3	1	0	1	1	0	1	0	0	1
e_4	0	1	0	0	1	0	1	1	0
e_5	1	0	0	0	1	1	0	1	1

TABLE 2.3 – Exemple de correspondance entre les instances, les types et les propriétés

L'approche construit à travers des requêtes un tableau comme celui décrit dans la table 2.3. Les lignes correspondent à des instances (e_i) et les colonnes correspondent à des classes (C_i), des propriétés entrantes (P_{di}) ou des propriétés sortantes (P_{ri}). Si la relation est vraie entre la ligne et la colonne, alors la valeur de leur intersection est de 1, sinon, elle est de 0. Un algorithme de déduction de règles d'association est ensuite appliqué sur cette table. Comme exemple de règles d'association qui peuvent être découvertes à partir de la table 2.3 :

1. $C_3 \rightarrow C_1$
2. $C_1 \rightarrow P_{d3}$
3. $C_2 \rightarrow P_{r1}$

A partir de chacune de ces règles d'association, l'ontologie est enrichie respectivement par les déclarations suivantes :

1. C_3 $rdfs{:}subClassOf$ C_1
2. P_3 $rdfs{:}domain$ C_1
3. P_1 $rdfs{:}range$ C_2

Des déclarations *rdf:type* doivent être fournies explicitement dans la source de données pour que l'approche puisse inférer d'autres informations sur le schéma. Plus précisément, l'approche utilise les règles d'association pour enrichir l'ontologie avec des déclarations RDFS (domain, range, subclasseOf, subPropertyOf) et OWL (SymetricProperty, TransitiveProperty).

Approche de Heiko Paulheim (2012) : Déduction de nouveaux types à partir de types existants [38].

Le travail présenté dans [38] suppose également que pour certaines instances, l'information sur le type est incomplète. La méthode proposée utilise les règles d'association [37], plus précisément une variante plus rapide de l'algorithme Apriori [39] pour compléter l'information des types pour les données. Les règles générées sont, par exemple de cette forme : « $yago{:}Singer110599806$ », « $yago{:}AmericanMusicians$ » => « $yago{:}AmericanSingers$ ». Selon les auteurs, environ 3 types supplémentaires sont rajoutés à une instance avec une précision de 85.6 %. Comme *SDType* [40], cette approche permet d'inférer plusieurs types pour une ressource mais n'enrichit pas les instances avec des types qui ne sont pas déjà présents dans la source de données.

Approche de Nuzzolese, Andrea Giovanni et al. (2012) : Inférence de types à travers l'analyse des liens de *Wikipedia* [41].

Les auteurs de [41] recherchent des types pour les instances non-typées de *DBpedia* en exploitant les types des ressources provenant de ressources liées à *DBpedia*. Ces ressources liées sont trouvées en analysant également les liens avec les pages de *Wikipedia*. Pour chaque ressource, les caractéristiques des types connexes sont construites, et l'algorithme de classification des plus proches voisins K-NN [42] est appliqué pour prédire les types d'une instance de *DBpedia*. Cependant, cette méthode est spécifique à *DBpedia* et ne peut pas traiter un jeu de données quelconque.

Approche de Zong, N. et al. (2012) : Découverte de la hiérarchie des types [43].

Zong, et al [43] utilisent également un algorithme de clustering hiérarchique, mais pour construire une hiérarchie des types déclarés dans le jeu de données. Les déclarations *rdf:type* des instances sont exploitées pour cela. Le modèle d'espace vectoriel, qui est une méthode pour représenter un objet comme un vecteur, est utilisé ainsi que le coefficient de Jaccard. La méthode ne découvre pas les types du jeu de données mais elle découvre uniquement les liens hiérarchiques entre les types sur la base des déclarations *rdf:type* des instances.

2.3.2 Approches d'enrichissement par analyse statistique

Les approches présentées dans cette section, utilisent l'analyse statistique de la distribution des types pour inférer de nouveaux types à des instances.

Approche de Heiko Paulheim et al. (2013) : Inférence de types dans des sources de données RDF bruitées [40].

La plupart des jeux de données ouverts, et tout spécialement les Linked Open Data (LOD) contiennent des données bruitées et incorrectes, ce qui rend la déduction du type de ces données par raisonnement difficile. *SDType* (Statistical Distribution of Types) [40] est une heuristique d'inférence de type utilisant les liens entre les instances. Cette approche est capable de traiter des données bruitées et incorrectes. C'est une approche de vote pondéré qui considère de nombreux liens, ce qui évite la propagation des erreurs des instances non pertinentes. Il existe des règles d'inférence de type qui sont déduite à partir des caractéristiques de RDFS :

- (x *rdf:type* t1) et (t1 *rdfs:subClassOf* t2) ⇒ x *rdf:type* t2
- (x R y) et (R *rdfs:domain* t) ⇒ x *rdf:type* t
- (x R y) et (R *rdfs:range* t) ⇒ y *rdf:type* t

Cependant, ces règles ne sont pas toujours valables : par exemple, si on applique ces règles sur la ressource *DBpedia:Germany*, seulement trois des 23 types retournés sont corrects. La liste des types retournés contient beaucoup de types erronés, par exemple : *award*, *city*, *sports team*, *mountain*, *stadium*, *person* et *military conflict*. Cela s'explique par le fait qu'il suffit d'un seul triplet non pertinent pour inférer un type erroné.

Au lieu de tirer parti de l'information à partir du schéma (range, domain, subClassOf) uniquement, *SDType* considère également l'utilisation du schéma à travers la pertinence d'une propriété pour un type, ce qui la rend robuste aux éléments non pertinents. Pour une ressource donnée r, la méthode applique d'abord les règles d'inférence de type déduites à partir des caractéristiques de RDFS. Puis, pour chaque type T inféré, son degré de confiance pour la ressource r est calculé, selon les propriétés p de r, en tenant compte du poids de la propriété p pour le type T. Le poids d'une propriété pour un type est calculé comme la distribution de la propriété par rapport à la distribution des types. Les types de

r retenus, sont les types dont la confiance est satisfaisante (supérieure à un seuil fixé).

Type	Subject (%)	Object (%)
owl:Thing	100.0	88.6
dbpedia-owl:Place	69.8	87.6
dbpedia-owl:PopulatedPlace	0.0	84.7
dbpedia-owl:ArchitecturalStructure	50.7	0.0
dbpedia-owl:Settlement	0.0	50.6
dbpedia-owl:Building	34.0	0.0
dbpedia-owl:Organization	29.1	0.0
dbpedia-owl:City	0.0	24.2
...	...	...

TABLE 2.4 – Distribution de la propriété *DBpedia – owl:location* dans *DBpedia*

De manière générale, pour une ressource donnée « r », *SDType* retrouve ses types comme suit :

1. Appliquer les règles d'inférence de type déduites à partir des caractéristiques de RDFS, vues précédemment ;
2. Pour chaque type « T » inféré, calculer sa confiance pour la ressource « r », selon les propriétés « p » de « r ». Cela, en tenant compte du poids de la propriété p pour le type T. Le poids d'une propriété pour un type est calculé comme la distribution de la propriété par rapport à la distribution des types. Cela est calculé à partir de la table 2.4 ;
3. Les types de « r » conservés sont ceux dont le degré de confiance est satisfaisant.

La table 2.4 montre la distribution de la propriété *DBpedia – owl:location* sur *DBpedia*. La première colonne de la table liste les types déclarés dans *DBpedia*, la deuxième colonne représente le pourcentage des instances d'un type qui apparaissent comme des sujets de la propriété *DBpedia – owl:location*. La troisième colonne représente le pourcentage des instances d'un type qui apparaissent comme objets de la propriété *DBpedia – owl:location*. Par exemple, 87.6% des instances de type *DBpedia – owl:place* apparaissent comme objet de la propriété *DBpedia – owl:location*.

C'est donc une méthode statistique qui infère les types d'une ressource en utilisant les règles d'inférences RDFS, et qui calcule la confiance d'un type pour une ressource. La contribution dans *SDType* est de proposer une approche qui permette d'évaluer la pertinence d'un type inféré pour une ressource en utilisant les règles d'inférence RDFS. En effet, les primitives : *rdf:type*, *rdfs:domaine*, *rdfs:range* et *rdfs:subClassOf* lui sont indispensables pour inférer les types. D'ailleurs, cette approche ne peut pas introduire de nouveaux types car tous les types qui seront affectés à une ressource doivent

être présents dans le jeu de données et déjà attribués auparavant à d'autres ressources dans le jeu de données par une déclaration *rdf*:*type*. En revanche, elle permet d'attribuer un poids aux propriétés selon leur pertinence pour inférer un type, mais cela nécessite qu'une partie du jeu de données ait un type qui va être utilisée comme ensemble d'apprentissage. La méthode est incrémentale mais elle ne peut pas être utilisée en l'absence totale de déclarations de types.

Approche de Lu Fang. et al. (2016) : Inférence de type en utilisant les catégories [44].

Cette approche propose d'identifier le type d'une instance en fonction de l'information sur sa catégorie fournie à travers la déclaration *dcterms*:*subject*. La répartition statistique de chaque catégorie est d'abord calculée sur tous les types. Ensuite, pour une instance donnée, des types candidats sont générés selon la probabilité de distribution de sa catégorie. Enfin, le type correct est identifié en fonction de la probabilité de distribution, des mots clés dans la catégorie et du résumé de l'instance.

L'approche est similaire à *SDType*, car elle est fondée sur la distribution statistique des types. La différence avec *SDType* est qu'elle infère le type d'une instance selon les informations de la catégorie de l'instance au lieu d'utiliser toutes ses propriétés. Les catégories sont choisies car elles sont prédictives pour le type d'une instance. En outre, puisque cette méthode ne considère que l'information sur la catégorie, elle est plus efficace pour une source de données de grande échelle tel que *DBpedia*.

2.3.3 Discussion sur les approches d'enrichissement du schéma explicite

Les approches [36, 40, 44, 38, 41, 43] qui enrichissent un schéma à partir des déclarations explicites fournies dans un jeu de données sont synthétisées dans le tableau 2.5.

Toutes les approches approches qui enrichissent le schéma existant nécessitent des déclarations spécifiques sur le schéma comme : *rdf*:*type* [36, 40, 44, 38, 43], *owl*:*sameAs* [41], *rdfs*:*domain* [36], *rdfs*:*range* [36], *rdfs*:*subClassOf* [36] ou la déclaration sur la catégorie *dcterm*:*subject* [44]. Ces approches enrichissent le schéma existant avec différentes déclarations supplémentaires : *rdf*:*type* [36, 40, 44, 38, 41], *rdfs*:*domain* [36], *rdfs*:*range* [36], *rdfs*:*subClassOf* [36, 43], *rdfs*:*subPropertyOf* [36], *owl*:*SymetricProperty* [36] ou *owl*:*TransitiveProperty* [36].

La plus part de ces approches nécessitent la spécification de paramètres tels que : le seuil de support et de confiance pour les règles d'association [36, 38], le nombre de voisins [41] et le seuil de confiance pour un type [40, 44]. Ces ap-

	Approche	Déclarations indispensables	Techniques	Résultat	Paramètres	Types externes	Remarques
Fouille de données	**Schema induction – Volker et al. [36]**	• rdf:type	• Règles d'association	• rdfs :domain • rdfs :range • rdfs :subClassOf • rdfs :subPropertyOf • owl :SymetricProperty • owl:TransitiveProperty	Seuils de support et de confiance des règles d'association	non	• Règles d'association couteuses • Construction d'une table de transaction pour la découverte des règles d'association
	Heiko et al. [38]	• rdf:type	• Règles d'association	• Déclarations "rdf :type" supplémentaires	Seuils de support et de confiance des règles d'association	non	
	Nuzzolese et al. [41]	• Ressources liées à DBpedia : owl :sameAs	• Analyse des liens vers Wikipedia • K-NN	• "rdf :type"	Le nombre de voisins K	Oui	Spécifique à DBpedia
	Zong et al. [43]	• rdf:type	• Clustering hiérarchique	• rdfs :subClassOf		non	
Statistique	**SDType – Heiko et al. [40]**	• rdf :type • rdfs :domain • rdfs :range • rdfs :subClassOf	• Règles d'inférence RDFS • Statistique sur la distribution des propriétés sur les types	• Déclarations "rdf :type" supplémentaires avec un degré de confiance	Seuil de confiance pour un type	non	[44] est moins couteux que [40] car il utilise uniquement la propriété de la catégorie. Cependant, celle-ci est souvent manquante ou incomplète.
	Using catégories - Lu Fang et al. [44]	• rdf:type • dcterm:subject	• Statistique sur la distribution de la catégorie sur les types	• Déclarations "rdf :type" supplémentaires avec un degré de confiance	Seuil de confiance pour un type	non	

TABLE 2.5 – Synthèse des approches d'enrichissement du schéma explicite

proches ne peuvent pas enrichir les données avec des types qui n'existent pas dans le jeu de données, sauf l'approche [41] qui exploite les types des ressources provenant de ressources liées à *DBpedia* avec la déclaration *owl:sameAs*. Cependant, cette méthode est spécifique à *DBpedia* et ne peut pas traiter un jeu de données quelconque.

2.4 Découverte des patterns structurels des instances

Certaines approches tentent de découvrir les types (classes) d'un jeu de données, tandis que d'autres essayent de découvrir des patterns structurels. Une approche de découverte de types sur le graphe de la figure 2.22 nous renseignerait sur le fait qu'il existe des *Etudiants* et des *Publications*, tandis qu'une approche de couverte de patterns indique qu'il y a des instances décrites par exemple, par les structures $V1 = \{\overrightarrow{name}, \overrightarrow{address}, \overleftarrow{hasAuthor}\}$ ou $V2 = \{\overrightarrow{name}, \overrightarrow{office}, \overrightarrow{speciality}, \overleftarrow{hasAuthor}, \overrightarrow{write}\}$. On peut considérer un ensemble de patterns comme les versions d'un type, comme dans notre exemple où les patterns *V1* et *V2* représentent les versions possibles d'une instance de type *Etudiant*.

Les instances d'un jeu de données sont décrites par des ensembles de propriétés différents. Des instances d'un même type peuvent également être décrites par un ensemble hétérogène de propriétés. Un ensemble de propriétés décrivant une instance représente le pattern structurel de cette instance.

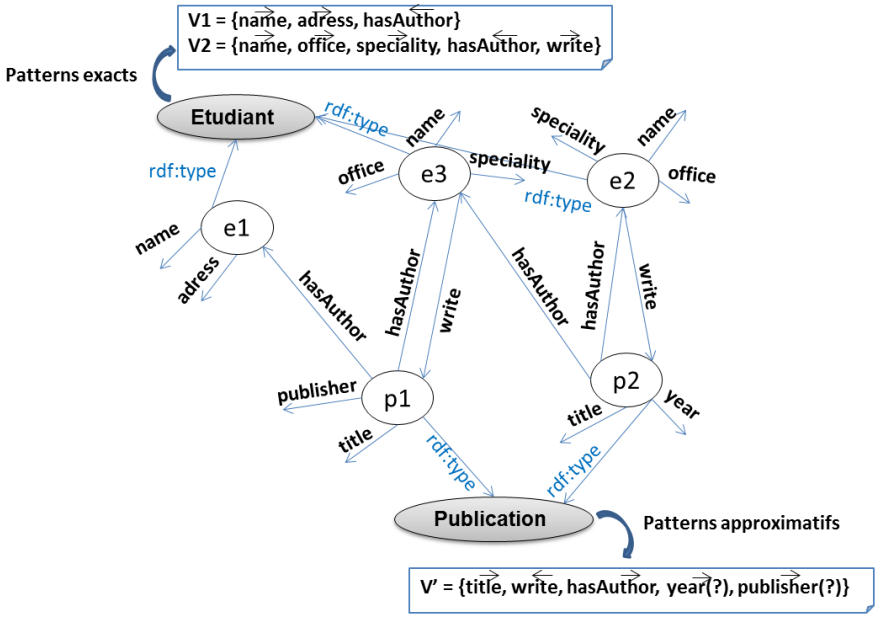


FIGURE 2.22 – Graphe RDF avec les patterns structurels des instances.

Un pattern structurel peut être exact ou approximatif. Un pattern exact représente la structure exacte d'un ensemble d'instances ; tandis qu'un pattern approximatif représente la structure approximative commune à un ensemble d'instances contenant éventuellement des propriétés optionnelles. Nous définissons un pattern exact et un pattern approximatif comme suit.

Définition (Pattern exact). Un pattern exact V d'un jeu de données est un ensemble de propriétés qui décrit une instance e du jeu de données, tel que :
- $\forall\ p \in V$: p décrit e ;
- $\forall\ p$ qui décrit e : $p \in V$.

Les patterns exacts d'un ensemble d'instances d'un type représentent les versions structurelles du type.

Définition (Pattern approximatif). Un pattern approximatif V' d'un jeu de données est un ensemble de propriétés qui décrit un ensemble d'instances similaires T du jeu de données, tel que :
- $\forall\ p$ décrivant toutes les instances dans T : $p \in V'$;
- Si p décrit certaines instances de T, p est optionnelle dans V'.

Il y a plusieurs façons de marquer une propriété p comme optionnelle dans un pattern, comme par exemple avec un (?) par référence à une propriété optionnelle dans les requêtes SPARQL.

La différence majeure entre un pattern exact et pattern approximatif, est que la co-occurrence des propriétés n'est pas renseignée dans un pattern approximatif. La figure 2.22, illustre la découverte des patterns exacts et approximatifs d'un jeu de données. En effet, les instances d'un type peuvent être décrites par un ensemble hétérogène de propriétés, comme les deux instances e_1 et e_2 de type *Etudiant*, et les deux instances p_1 et p_2 de type *Publication*. Nous donnons dans la figure 2.22, un exemple de patterns exacts pour les instances de type *Etudiant*. Nous donnons également un exemple de pattern approximatif pour les instances de type *Publication*. Nous pouvons voir que la co-occurrence des propriétés est parfaitement renseignée dans un pattern exact par le fait que chaque version d'un ensemble d'instances similaires (de même type dans notre contexte) est représentée par un pattern. Tandis que le pattern approximatif des instances de type *Publication*, ne renseigne pas sur la co-occurrence des propriétés. En effet, on ne sait pas s'il y a des instances, dans le jeu de données, qui sont décrites par les propriétés *year* et *publisher* en même temps.

Nous présentons dans la section 2.4.1, les approches qui découvrent les patterns structurels exacts d'un jeu de données. La section 2.4.2 présente les approches qui découvrent les patterns structurels approximatifs d'un jeu de données. Une discussion sur ces approches est présentée dans la section 2.4.3.

2.4.1 Approches de découverte de patterns exacts

Dans cette section, nous présentons les approches qui découvrent les patterns structurels exacts d'un jeu de données.

Approche de D. Ruiz et al. (2015) : Inférence d'un schéma versionné [6].

Cette approche découvre les différentes versions d'un schéma de données NoSQL exprimées en Json. Les versions d'un schéma représentent les différentes versions des types du jeu de données. En effet, les instances d'un type de données NoSQL peuvent avoir des structures différentes, qui représentent les versions de ce type.

Afin de découvrir les versions d'un schéma, les données sont partagées avec map/reduce sur la base de leur déclaration de type, de telle sorte que chaque nœud contienne uniquement des données de même type (chaque nœud représente un type dans le cas de données RDF). L'approche considère la déclaration de type dans un fichier Json comme une propriété type déclarée pour un objet. La problématique où cette déclaration est manquante n'est pas abordée, ni celle

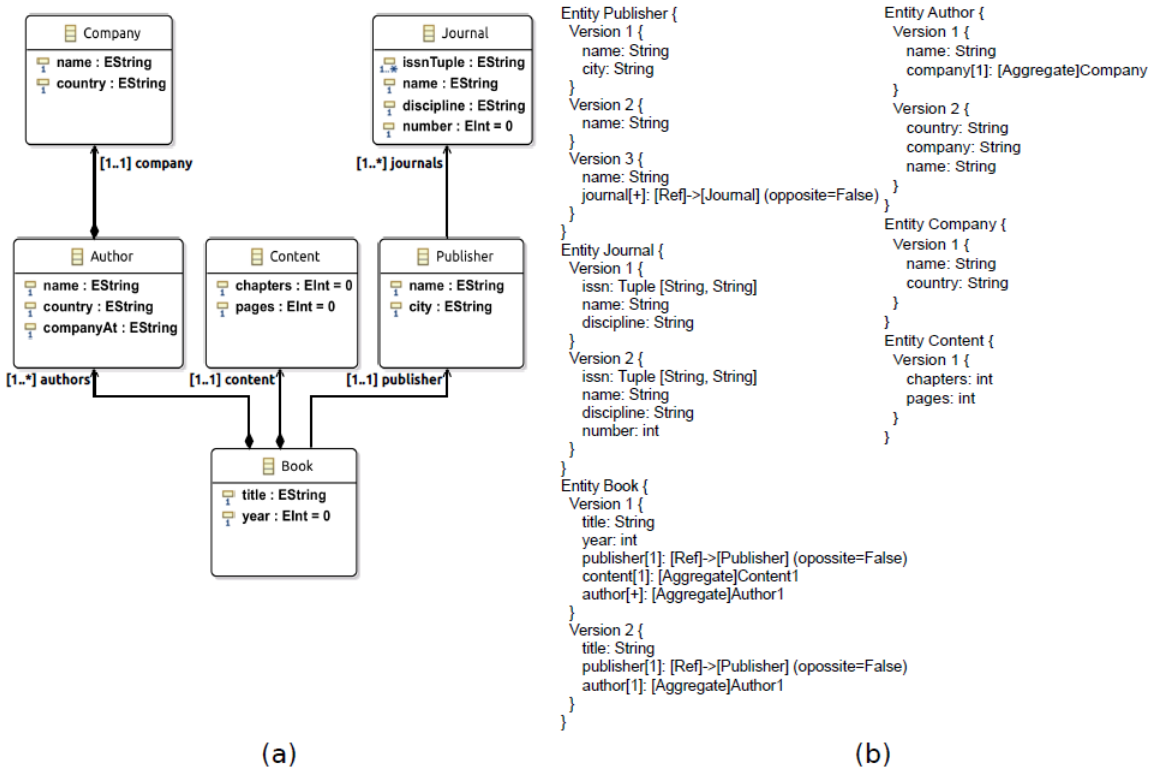


FIGURE 2.23 – Représentation graphique d'un schéma (a) avec la description textuelle des versions des types (b) [6].

où un objet possède plusieurs types. Pour découvrir les versions d'un type, les instances d'un nœud sont parcourues et chaque structure d'instance différente va représenter une version du type.

En plus de découvrir les différentes versions d'un type, un méta-modèle d'un schéma NoSQL est généré avec le type des valeurs de chaque propriété (chaîne de caractères, entier, etc). La figure 2.24 est un exemple de représentation graphique d'un schéma (a) avec la description textuelle des versions des types (b). Nous pouvons voir par exemple qu'une instance de *Publisher*, représentée par un seul type dans le schéma (voir figure 2.24 (a)), a en réalité trois versions dans le jeu de données, qui sont décrites textuellement dans la figure 2.24 (b). Notons que pour cette approche un pattern décrit uniquement les propriétés sortantes d'un type.

Approche de Belghaouti, Fethi et al. (2016) : FreGraPaD - Détection des patterns fréquents pour des données RDF en flux [7].

FreGraPaD [7] est un algorithme qui détecte en une seule passe les patterns fréquents dans les flux de données RDF. Cela, afin de les compresser.

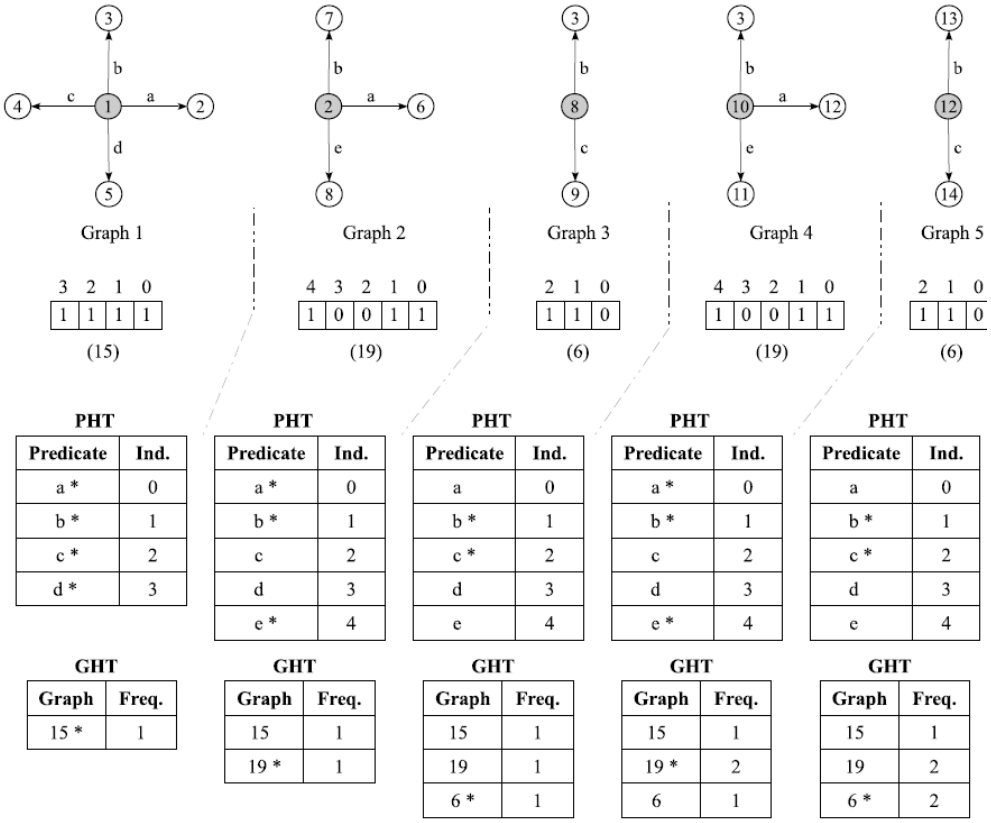


FIGURE 2.24 – Exemple de flux de données RDF : détection de patterns de graphe fréquents avec l'algorithme FreGraPaD [7].

Pour relever ces défis, FreGraPaD repose principalement sur deux structures de données : (i) un vecteur binaire Bit Vector PHT, afin d'identifier les patterns d'un graphe RDF et d'optimiser l'utilisation de l'espace mémoire ; (ii) un tableau de hachage GHT, pour identifier, détecter et maintenir les prédicats et les patterns.

La figure 2.24 illustre la façon dont FreGraPaD détecte des patterns sur un exemple de 5 instances RDF. La figure montre aussi l'évolution du vecteur de bits, des tables PHT et GHT à chaque fois qu'une instance est traitée. A la réception de la première instance (Graphe 1), l'algorithme vérifie un par un la présence de ses prédicats a, b, c et d dans PHT. Chaque bit correspondant dans le vecteur de bits est alors mis à 1 et l'indice augmenté respectivement de 0 à 1, 2 et 3, car tous sont de nouveaux prédicats. On obtient donc 1111 (15) comme valeur pour le vecteur bit. Ainsi, chaque prédicat est inséré dans PHT avec son indice correspondant <a ; 0>, <b ; 1>, <c ; 2> et <d ; 3>. Après traitement du dernier prédicat (d), l'algorithme vérifie la présence du vecteur de bit construit (constituant le graphe graphique 1111 (15)) dans GHT. Si le pattern correspondant existe alors sa fréquence est incrémentée sinon, il est inséré avec la fréquence

1, c'est-à-dire <15 ; 1>.

FreGraPaD est un algorithme d'extraction de patterns d'un flux de données RDF. Cependant, il ne considère pas les propriétés entrantes d'une instance. Il est fondé sur le comptage des patterns des instances. A chaque passage d'une instance, les tables des prédicats PHT et la table des patterns GHT sont parcourues et mises à jours.

Approche de M. Konrath et al (2012) : Schemex - Construction d'un catalogue par indexation sur des sources de données distribuées [8].

Pour trouver les sources de données pertinentes pour une requête donnée, SchemEX [8] construit une structure d'index de triplet RDF sur des sources de données distribuées. Les déclarations *rdf:type* sont utilisées pour trouver les classes, ensuite les classes équivalentes sont retrouvées sur les différentes sources.

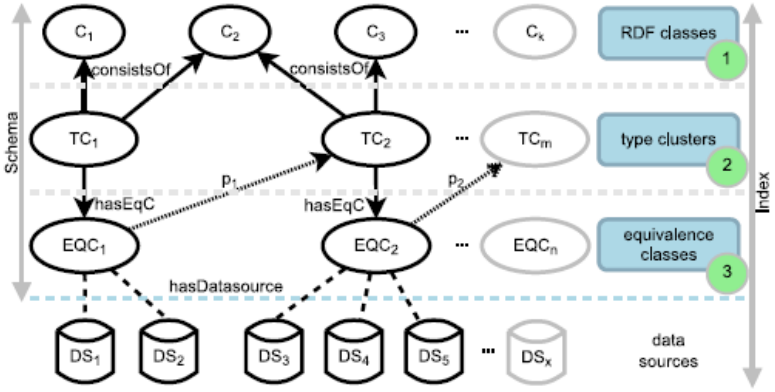


FIGURE 2.25 – Structure de l'index renforcée avec deux couches supplémentaires tirant parti des déclarations *rdf:type* [8]

Comme le montre la figure 2.25, l'index construit par SchemEX est composé de trois couches :

- Classes RDF : permettent de mapper chaque instance avec le type de sa classe (déduit grâce aux déclarations *rdf:type*) ;
- Clusters de type : regroupe les classes qui ont des instances en commun. Cela permet de sélectionner les sources des instances qui sont à la fois de type *Politicien* et *Acteur*, par exemple ;
- Classes d'équivalence : une classe d'équivalence est constituée d'instances ayant les mêmes propriétés, ce qui représente dans notre contexte un pattern structurel exact. Elle est caractérisée par la structure d'une instance, exemple : nom, prénom, âge, job, etc. Chaque cluster de type a plusieurs classes d'équivalence.

La méthode est incrémentale, car à l'arrivée d'une nouvelle instance, on détermine : (i) ses classes d'appartenance, et ce grâce à la primitive *rdf:type*, (ii) son cluster d'appartenance qui est le cluster qui regroupe ses classes d'appartenance, et (iii) sa classe d'équivalence qui est la structure (pattern exact) qui représente l'ensemble des propriétés qui la décrivent.

2.4.2 Approches de découverte de patterns approximatifs

Dans cette section, nous présentons les approches qui découvrent les patterns structurels approximatifs d'un jeu de données. La différence principale avec les approches de découverte de patterns structurels exacts est que les patterns approximatifs ne renseignent pas sur la co-occurrence entre les propriétés.

Approche de Baazizi et al. (2017) : Inférence de schéma pour des jeux de données massifs exprimés en JSON [45].

L'approche proposée permet de caractériser un jeu de données exprimé en Json par les différentes structures de ces données et d'identifier les types des valeurs de leurs propriétés (string, number, etc). Une structure de données est décrite par une expression régulière, avec la spécification du type de chaque propriété. Le paradigme de *Map/reduce* est utilisé pour le passage à l'échelle. L'approche procède en deux étapes :

1. Inférence des types (*Map*) : on considère ici un type, non pas comme une déclaration *rdf:type* en RDF, mais comme la structure d'une instance avec le type de chaque propriété (string, number, etc). La première étape de l'approche est faite avec *Map* : les mappeurs examinent tous les objets et, pour chaque objet, renvoient un enregistrement contenant un type (structure) inféré comme clé et, dans WordCount, la valeur 1. Par exemple, soit un jeu de données contenant les quatre objets ci-dessous :
   - Person { id : 1, age : 14, admin : false, name : "John Smith", phone : 3132437}
   - Person { id : 2, name : "Edmond Dantes", email : "ed@mc.com", admin : true}
   - Office { id : 3, number : 3, address : "4 rue armengaud"}
   - Person { id : 4, name : "Amanda Clarke", age : 26, admin : false, phone : 2123142222}

   Un type est inféré pour chaque objet durant la phase *Map* de la façon suivante :
   - T_1 = { id : Number, age : Number, admin : Bool, name : String, phone : Number}
   - T_2 = { id : Number, name : String, email : "ed@mc.com", admin : Bool}
   - T_3 = { id : Number, number : Number, address : String}

- T_4 = { id : Number, name : String, age : Number, admin : Bool, phone : Number}

Les enregistrements <Clés, WordCount> sont les suivants : <T_1 ; 1>, <T_2 ; 1>, <T_3 ; 1> et <T_4 ; 1>. Avant que la phase de réduction ne commence, les enregistrements produits par les *mappeurs* sont regroupés en comparant leurs valeurs de clés au moyen d'un algorithme d'appariement de type. Sur l'exemple précédent, l'algorithme découvre que T_1 et T_4 sont équivalents ; par conséquent, la phase de réduction prend comme entrée les paires <T_1 ; {1, 1}>, <T_2 ; 1>, et <T_3 ; 1>.

2. Fusion des types (Reduce) : la fusion de types est la deuxième étape de l'approche et consiste à fusionner itérativement les types produits lors de la phase *Map*. La fusion de type est faite de façon distribuée, elle repose sur un opérateur de fusion commutatif et associatif. Cet opérateur de fusion est invoqué sur deux types T_1 et T_2, et produit un super type structurel. L'idée principale est de ne représenter qu'une fois ce qui est commun et, en même temps, de préserver toutes les parties différentes qui sont propres à chaque type. Les parties communes sont fusionnées et les parties différentes sont considérées comme facultatives dans le type résultant et elles sont suivies par un point d'interrogation « ? » ; par exemple T_1 et T_2 sont fusionnés comme suit :
   - $T_{1,2}$ = { id : Number, (age : Number) ?, admin : Bool, name : String, (phone : Number) ?, (email : String) ?}

   Si deux structure T et U ne sont pas de même nature (dans le sens *rdf:type*), la fusion produit une simple union $T + U$; par exemple, la fusion de $T_{1,2}$ avec T_3 produit :
   - $T_{1,2,3} = T_{1,2} + T_3$ = {id : Number, (age : Number) ?, admin : Bool, name : String, (phone : Number) ?, (email : String) ?} + {id : Number, number : Number, address : String}

   La fonction de fusion doit traiter le cas où les types $T = T_1 + ... + T_n$ et $U = U_1 + ... + U_m$ doivent être fusionnés. Dans ce cas, la fonction de fusion identifie et fusionne les types T_j et U_h de même nature (*rdf:type*), alors que les structures de nature différentes sont simplement déplacées dans le type union de sortie.

Le schéma inféré par cette approche est composé des différentes structures de données décrites sous-forme d'expressions régulières. Une expression régulière est composée des propriétés de la structure. Une propriété dans une structure peut être obligatoire si elle est définie pour toutes les instances de la structure ou optionnelle, et dans ce cas annotée par un « ? ». Si une propriété est définie plusieurs fois pour une instance, elle est annotée par *. Dans ces expressions régulières, le type de valeur de chaque propriété est également décrit.

L'inférence de schéma de cette approche consiste en la découverte des structures possibles des données ainsi que le type des valeurs des propriétés et les décrire avec des expressions régulières. L'approche ne découvre pas les types des données mais les types de la valeur d'une propriété comme : string, number, etc. L'information sur le type (*kind*) d'un objet json est considérée comme fournie dans le jeu de données. La problématique de la découverte des liens entre les types ou les structures inférées n'est pas abordée.

Approche de Zneika, Mussab et al. (2016) : Résumer un graphe RDF par la découverte de patterns approximatifs [46].

L'approche proposée dans [46], et présentée plus en détails dans [9], découvre les top-K patterns approximatifs d'un graphe RDF. Chaque pattern découvert est également instancié par le nombre d'instances qui le suivent. Ce qui permet à une requête d'estimer les résultats attendus. De cette façon, il est possible d'interroger le résumé du graphe RDF afin d'identifier si les informations nécessaires sont présentes et si elles le sont en nombre significatif pour être incluses dans un résultat de requête.

Les auteurs proposent de trouver les top-k patterns approximatifs d'un jeu de données, en appliquant un algorithme de pattern mining (PANDA [47]). Cet algorithme extrait les patterns de façon itérative, et chaque pattern est construit progressivement en ajoutant de nouveaux éléments et en vérifiant les lignes de la matrice qui correspondent approximativement à ces éléments avec un seuil de bruit qui tolère des faux positifs.

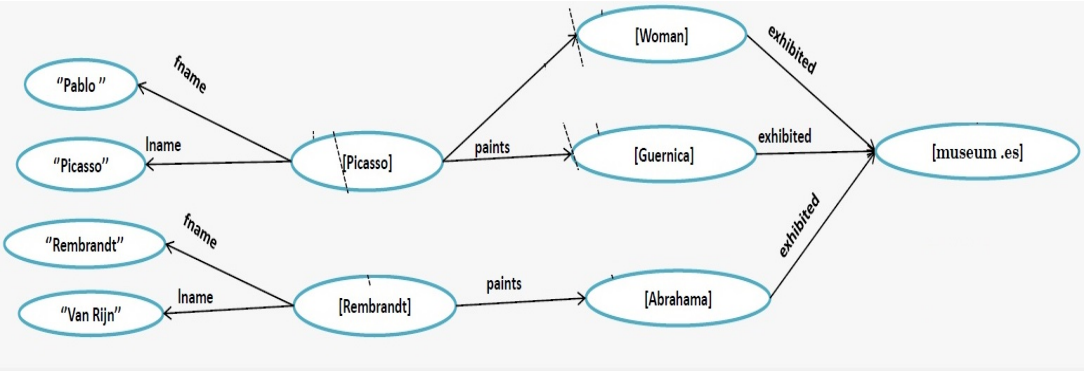


FIGURE 2.26 – Exemple de graphe RDF [9].

Comme PANDA s'applique sur une matrice binaire, un graphe RDF est d'abord transformé en une matrice binaire comme suit :

- Les lignes représentent les sujets (instances) ;
- Les colonnes représentent les prédicats et les types si la déclaration *rdf:type* est présente. Un prédicat est considéré comme une propriété si son objet

est une instance. Un prédicat est considéré comme un attribut si son objet est un littéral ;
- Afin de capturer à la fois le sujet et l'objet d'une propriété, deux colonnes pour chaque propriété sont créées. La première colonne capture l'instance qui est le sujet (qui appartient au domaine de la propriété), tandis que la seconde, appelée propriété inversée, capture l'instance qui est l'objet (qui appartient au co-domaine de la propriété) ;
- La valeur à la ligne i de la colonne j de la matrice $D(i, j) = 1$ si i a le prédicat j ou si i a la propriété inversée j ou si i est du type de j (selon si j est un prédicat, une propriété inversée ou un type) ; sinon $D(i, j) = 0$.

Soit le graphe RDF de la figure 2.26. La table de la figure 2.27 représente la matrice binaire de ce graphe. Cette matrice est composée de 9 colonnes et 6 lignes, où les colonnes représentent 2 attributs distincts *(fname, lname)*, 2 propriétés distinctes *(paints, exhibited)*, 2 propriétés distinctes inversées *(R_paints, R_exhibited)* et 3 types *(Painter (c), Painting (c), Museum (c))*. Afin de distinguer les types et les propriétés/attributs au niveau de la visualisation, *Y (c)* est utilisé pour indiquer que Y est un type. Les lignes représentent les 6 instances distinctes *(Picasso, Rembrant, Woman, Guernica, Abraham, museum.es)*. Par exemple, $D(1,3) = 1$ car *Piccaso* a l'attribut *lname*, et $D(1,6) = 0$ car il n'a pas la propriété *exhibited*.

	Painter(c)	Painting(c)	lname	fname	paints	exhibited	R_paints	R_exhibited	Museum(c)
Picasso	1	0	1	1	1	0	0	0	0
Rembrant	1	0	1	1	1	0	0	0	0
Woman	0	1	0	0	0	0	1	0	0
Guernica	0	1	0	0	0	1	1	0	0
Abraham	0	1	0	0	0	1	1	0	0
museum.es	0	0	0	0	0	0	0	1	1

FIGURE 2.27 – Matrice binaire du graphe RDF de la figure 2.26.

Après la construction de la matrice binaire, l'algorithme de pattern mining PANDA est appliqué. La table de la figure 2.28 montre les patterns possibles qui peuvent être extraits de la matrice binaire de la figure 2.27. La première colonne représente l'identifiant du pattern. La seconde colonne représente les prédicats inclus dans un pattern et la troisième colonne représente le nombre d'instances par pattern, par exemple, le pattern $P1$ indique qu'il y a trois instances appartenant au type *Painting* et ayant *exhibited* comme une propriété sortante et *paint* comme une propriété entrante. Il convient de noter, qu'une instance assignée à un pattern ne doit pas nécessairement avoir tous ses prédicats.

Un schéma est généré à partir des patterns approximatifs comme suit :
- Un nœud est généré pour chaque pattern.
- Un arc est généré entre le nœud ayant une propriété et le nœud ayant la propriété inversée.

ID	Pattern	Correspondence class
P1	Painting(c),exhibited, revers_paint	3
P2	Painter(c),paints, fname, lname	2
P3	Museum(c)	1

FIGURE 2.28 – Exemple de patterns approximatifs extraits de la figure 2.27.

- Un arc est généré entre un nœud ayant un attribut et une URI représentant le type des valeurs de l'attribut.
- Un nœud est lié à son type avec un arc étiqueté *rdf:type*.
- Un arc étiqueté *extent* est également généré pour chaque nœud pour préciser le nombre d'instances que représente le nœud.

Dans ce travail, des patterns approximatifs sont retrouvés. L'approche nécessite la construction d'une matrice binaire de taille N x M, avec N le nombre d'instances du jeu de données et M représente le nombre de : prédicats (propriété et attribut), propriétés inversées et types si présents. La taille de cette matrice est très grande et elle dépasse même la taille du jeu de données, ce qui nécessite une grande capacité de mémoire.

L'approche tente de trouver le centroide (means) de chaque type (sans forcément considérer la déclaration *rdf:type*). Chaque type est caractérisé par un pattern approximatif. L'approche ne découvre pas toutes les versions d'un type mais un pattern approximatif pour chacun. Ce qui implique une perte d'information au niveau du schéma généré. Les expérimentations montrent également que pour une type on peut avoir un, zéro ou plusieurs patterns approximatifs. Le cas où une instance possède plusieurs types n'est pas abordé.

Approche de Cebiric S. et al. (2015) : Construction d'un filtre pour les requêtes d'un graphe RDF [10].

L'approche proposée dans [10] vise à résumer un graphe de données RDF en inférant une représentation qui n'est pas à proprement parler un schéma des instances décrites, mais un résumé permettant de déterminer si le jeu de données peut contenir la réponse à une requête. L'approche exploite les informations implicites sur le schéma, c'est à dire, les déclarations *rdfs:domain*, *rdf:type*, *rdfs:range* qui ne sont pas formellement fournies dans le jeu de données, mais qui peuvent en être déduites, par exemple : (a *rdf:type* c1 . c1 *rdfs:subClassOf* c2) $\Rightarrow$ (a *rdf:type* c2). Les étapes principales de l'approche sont les suivantes :

1. Saturer le graphe de données de telle sorte que toutes les informations implicites soient explicitement formulées. Cela, en utilisant par exemple des règles d'inférence RDFS comme pour la déduction des déclarations *rdf:type* à partir des déclarations *rdfs:subclassOf* ;
2. Les nœuds du schéma Gs sont formés sur la base du domaine/co-domaine

des propriétés, non pas par rapport aux déclarations "*rdfs:domain/range*", mais selon la position dans un triplet, comme suit : soient le graphe de données G, et p1, p2 des propriétés ; S(p) représentant un domaine dans le graphe de schéma Gs et T(p) représentant un co-domaine dans Gs :
- Si (s p1 o1, s p2 o) ∈ G, alors S(p1) = S(p2) dans Gs
- Si (s1 p1 o, s2 p2 o) ∈ G, alors T(p1) = T(p2) dans Gs
- Si (s p1 o1, o1 p2 o2) ∈ G, alors T(p1) = S(p2) dans Gs

Les triplets composés d'une déclaration *rdf:type* dans le graphe des données G, sont traités dans le graphe schéma Gs comme suit :
- Si (s p o, s *rdf:type* c) ∈ G, alors S(p) T c ∈ Gs
- Si (s p o, o *rdf:type* c) ∈ G, alors T(p) T c ∈ Gs
- Si (s *rdf:type* c) ∈ G, alors ajouter un nouveau nœud de type c dans le schéma : n T c ∈ Gs.

3. Les déclarations *rdf:type* sont considérées pour ne pas regrouper des domaines de types différents, par exemple, "zipcode" peut avoir comme domaine "Person" et "store".
4. Un nœud annoté S(...) est créé dans le schéma pour représenter le domaine de chaque ensemble de propriétés dont le types des instances n'est pas déclaré.

Prenons comme exemple le graphe de données G décrivant les ressources r1, r2 et r3, et contenant les triplets suivants :
- <r1 rdf :type Book> ; <r1 hasTitle "T1"> ; <r1 hasAuthor "A1"> ; <r1 hasAuthor "A2">
- <r2 rdf :type EnPub> ; <r2 rdf :type Article> ; <r2 hasTitle "T2"> ; <r2 hasAuthor "A3"> ; <r2 hasReview "R1">
- <r3 hasTitle "T3"> ; <r3 hasAuthor "A4">

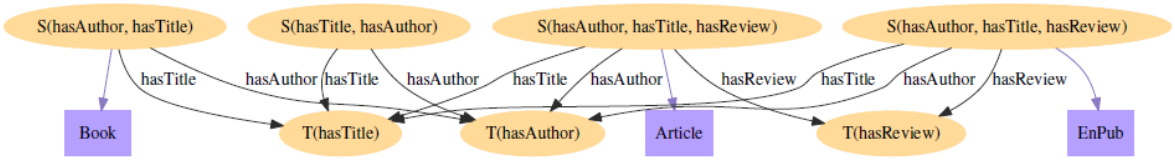


FIGURE 2.29 – Schéma extrait à partir des données du graphe G [10].

La figure 2.29 montre le schéma Gs extrait des données du graphe G. Chaque nœud annoté S(...) dans le schéma Gs représente le domaine d'un ensemble de propriétés et fait référence à un type unique. Chaque nœud annoté T(p) dans le schéma représente le co-domaine d'une propriété p. Chaque rectangle dans le schéma représente un type. Dans Gs, quatre URI ont été créées en haut de la figure 2.29 pour représenter les ressources ayant à la fois un titre et un auteur, respectivement (de gauche à droite) : les ressources de type *Book*, celles n'ayant

aucun type, les ressources de type *Article*, et celles ayant le type *EnPub*. L'URI de bas la plus à droite au bas de la figure 2.29 montre que seules les ressources de type *Article* ou *EnPub* ont des *reviews*.

L'idée générale de cette approche est le regroupement des propriétés selon le type de leur sujet dans les triplets, en prenant en considération les déclarations *rdf:type* pour ne pas regrouper deux types différents. Si pour certains, le type n'est pas déclaré, un nouveau nœud est créé. On peut constater que les nœuds domaine S(...) dans le schéma, sont liés à un type si la déclaration de type est fournie dans le jeu de données. Ces nœuds peuvent être comparés avec le pattern d'un type en terme de propriétés sortantes d'une instance de ce type. Cependant, la co-occurrence entre les propriétés d'un type n'est pas renseignée dans le résumé. De plus, selon les auteurs l'approche est très coûteuse avec une complexité de $O(|G|^5)$, où |G| représente le nombre de triplets dans la source de données décrivant le graphe de données. L'approche fournit un filtre pour les requêtes qui permet de savoir si la réponse à une requête se trouve dans la source de données. Cependant, une réponse positive ne garantit pas la présence de la réponse dans la source, car l'index ne renseigne pas sur la co-occurrence des propriétés ni sur leur valeurs.

2.4.3 Discussion sur les approches de découverte de patterns structurels

Les approches qui découvrent les patterns structurels exacts [6, 7, 8] ou approximatifs [45, 46, 10] à partir d'un jeu de données sont synthétisées dans le tableau 2.6.

Les approches présentées pour la découverte de patterns structurels d'un jeu de données ne considèrent que les propriétés sortantes des instances. Cependant, les propriétés entrantes sont aussi importantes pour caractériser une structure. Ces approches sont proposées pour des sources de données Json en local [6, 45], des sources de données RDF en local [46, 10], des données RDF en flux [7] ou des sources de données RDF distribuées [8]. Toutes ces approches nécessitent de parcourir les données pour trouver les patterns, ce qui rend leur utilisation impossible sur des sources de données distantes.

Dans un pattern approximatif, la co-occurrence des propriétés n'est pas renseignée. En effet, un pattern approximatif représente un ensemble d'instances similaires structurellement et pas forcément identique structurellement. L'approche proposée dans [45] marque les propriété optionnelles d'un pattern, tandis que l'approche proposée dans [10] ne les marque pas, et pour celle proposée dans [46], les propriétés optionnelles ne font pas partie de la description du pattern.

Approches		Déclarations indispensables	Techniques	Description d'un pattern	Résultat	Type de données
Patterns exacts	**Ruiz et al. [6]**	Le type d'un objet	• Division sur la base du type avec Map/Reduce • Parcours des données	• Propriétés sortantes uniquement	• Schéma E/R • Description textuelle des patterns de chaque type	Json en local
	FreGraPad – Belghaouti et al. [7]		• Parcours et comptage des instances • Représentation d'une structure par un vecteur de bits • Table de hachage	• Propriétés sortantes uniquement	• Patterns exacts avec le nombre d'occurrence de chaque	Flux de données RDF
	SchemEx - Konrath et al. [8]	rdf:type	• Parcours des instances et construction de trois couches d'index	• Propriétés sortantes uniquement	• Index à trois couches • L'une des couches (classe d'équivalence) représente les patterns exacts	Sources de données RDF distribuées, avec accès non restreint
Patterns approximatifs	**Bazzizi et al. [45]**	le type (kind) d'un objet	• Division aléatoire avec map/reduce car opération de fusion associative et commutative	• Propriétés sortantes uniquement • Propriétés optionnelles marquées par "?"	• Expression régulière décrivant les patterns approximatifs découverts	Json en local
	Zneika et al. [46]	rdf:type exploité si fourni	• Construction d'une matrice binaire de transaction pour toutes les instances • Algorithme de pattern mining Panda	• Propriétés sortantes uniquement • Les propriétés optionnelles ne font pas partie du pattern	• Schéma approximatif inféré à partir des patterns approximatifs avec le nombre d'occurrence de chaque pattern (déclaration « extent »)	RDF en local
	Ceberic et al. [10]	rdf:type exploité si fourni	• Saturation du graphe de données • Regroupement des propriétés selon le type de leur sujet dans les triplets, tout en considérant les déclarations rdf :type	• Propriétés sortantes uniquement • Les propriétés optionnelles font partie du pattern sans être marquées	• Filtre pour les requêtes composé de patterns approximatifs • Le filtre tolère les faux positif vu que la co-occurrence entre les propriétés n'est pas renseignée ni leurs valeurs	RDF en local

TABLE 2.6 – Synthèse des approches de découverte de patterns structurels des instances

2.5 Annotation des données

Certaines des approches présentées précédemment pour la découverte du schéma implicite de la source de données forment des clusters d'instances similaires qui représentent des types. Cependant, il reste à découvrir le sens de ces clusters ?

La figure 2.30 représente des groupes d'instances similaires structurellement. L'annotation permet entre autres de capturer le sens de ces groupes ; il s'agit de trouver les termes permettant de déterminer que le cluster C_1 est un groupe d'*Etudiant* et que le cluster C_2 est un groupe de *Publication*. Notons que l'annotation peut aussi être utilisée pour documenter des types déclarés dans la source de données.

L'annotation consiste à étiqueter des objets ou un groupe d'objets afin d'expliquer leur sémantique. Plusieurs approches d'annotation ont été proposées dans des contextes différents pour : des données liées, des tables Web, des documents et du texte. Nous présentons dans ce qui suit des approches d'annotations dans

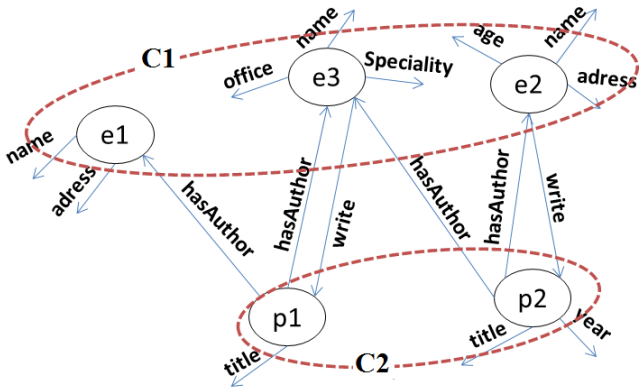


FIGURE 2.30 – Exemple de clusters d'instances similaires d'une source de données

ces différents contextes.

2.5.1 Approches d'annotation de données liées

Nous avons présenté dans la section 2.2.1 les approches de découverte du schéma implicite par regroupement des instances. Ces approches ne proposent généralement pas une méthode pour trouver des termes caractérisant les groupes formés. Cependant, certaines d'entre elles annotent ces groupes par des déclarations fournies dans la source de données. Dans cette section, nous présentons la façon dont ces approches annotent les données liées.

Approche de Chen, Jesse Xi et Reformat, Marek (2014) : Annotation des catégories découvertes [2].

L'approche proposée dans [2] applique un algorithme de clustering hiérarchique sur les données, où à chaque itération, les clusters les plus similaires sont regroupés. Une hiérarchie de catégories est alors construite. Le degré d'appartenance de chaque instance à une catégorie est également évalué.

Chaque cluster est annoté par sa catégorie, c'est-à-dire qu'il est étiqueté par un nom qui représente son sens. Ces noms sont retrouvés en tenant compte de deux types de déclarations dans la source de données :

- <subject *dcterm:subject* object>
- <subject *rdf:type* object>

La déclaration *dcterm:subject* fait partie du vocabulaire DCMI (Dublin Core Metadata Initiative), qui est un vocabulaire de métadonnées. Dans un triplet où la propriété est *dcterm:subject*, l'objet est généralement représenté à l'aide de mots-clés ou de phrases-clés pour décrire la nature de l'instance.

Les valeurs identiques de ces déclarations sont identifiées pour toutes les instances définies dans un cluster. Ces valeurs vont représenter les étiquettes du

C5:		
Thing, Feature, Place, Populated_Place, Administrative_District, Physical_Entity		
Region, YagoGeoEntity, Location_Underspecified		
France(0.95), **UK**(1.00), **Japan**(0.88), **US**(0.86), **London**(0.69), **Edinburgh**(0.67)		
C4:		
Member_states_of_the_United_Nations, G20_nations		
Liberal_democracies, G8_nations		
France(1.00), **UK**(1.00), **Japan**(0.91), **US**(0.86)		
C1:	**C3:**	
Member_states_of_the_EU	Countries	**C2:**
Countries_in_Europe	Bordering	British_capitals
Western_Europe	ThePacific	Capitals_in_Europe
Member_states_of_NATO	Ocean	Settlement, City
Countries_bordering_the_Atlantic		
France(1.00), **UK**(1.00)	**Japan**(1.00), **US**(1.00)	**London**(1.00), **Edinburgh**(1.00)

FIGURE 2.31 – Annotation des clusters découverts de la figure 2.7, et précision du degré d'appartenance de chaque instance [2]

cluster. Ce processus est répété pour tous les clusters. Le concept le plus général (au sommet de l'arbre hiérarchique) est d'abord annoté, puis les clusters qui sont plus loin dans la hiérarchie en ajoutant plus d'étiquettes. Les catégories des clusters feuilles de l'arbre hiérarchique sont les plus spécifiques, elles possèdent le plus grand nombre d'étiquettes. La figure 2.31 montre l'annotation des clusters découverts de la figure 2.7, et le degré de confiance qui capture la similarité de l'instance avec le centre du cluster.

Cette approche [2] introduit une méthodologie qui permet de découvrir les catégories de données RDF. Les déclarations sur le type des données ne sont pas utilisées pour la découverte des clusters mais uniquement pour les annoter. L'annotation se base sur les déclarations *rdf:type* et les déclarations *dcterm:subject*, de sorte à ce que, si une déclaration est la même pour toutes les instances d'un cluster, alors elle sera considérée comme un label du cluster. Cependant, ces déclarations sont souvent incomplètes et voir même inexistantes.

Approche de Christodoulou, K. et al. (2013) et de de S. Nestorov et al. (1997 - 1998) : Annotation des types découverts [26, 3, 31].

Certaines approches de découverte de type proposent de les annoter. La plupart de ces approches ne fournissent pas une méthode spécifique pour produire le nom pour les types générés. L'approche proposée dans [26], annote un type découvert par la déclaration *rdf:type* la plus fréquente au niveau des déclarations des instances formant le groupe de ce type.

Les approches d'extraction de schéma de données OEM proposées dans [3, 31], annotent un groupe d'instances de même type par le label le plus fréquent des arcs entrants. Ce qui équivaut à considérer la déclaration *rdf:type* la plus fréquente

dans un cluster d'instances, dans le contexte de données RDF. Cette approche est similaire à celle proposée dans [26].

2.5.2 Approches d'annotation de tables Web

Un autre contexte dans lequel l'annotation a été étudiée est celui des tables Web qui sont vues comme des données tabulaires. En effet, le Web offre un corpus de plus de 100 millions de tables, mais celles-ci sont rarement explicites et les lignes d'en-tête ne sont pas toujours présentes. Afin de comprendre le sens de chaque colonne, certaines approches annotent les données tabulaires, comme décrit dans la figure 2.32. Les annotations ajoutées sont les noms et les types des colonnes, les types des lignes et les relations binaires entre les colonnes d'une table Web.

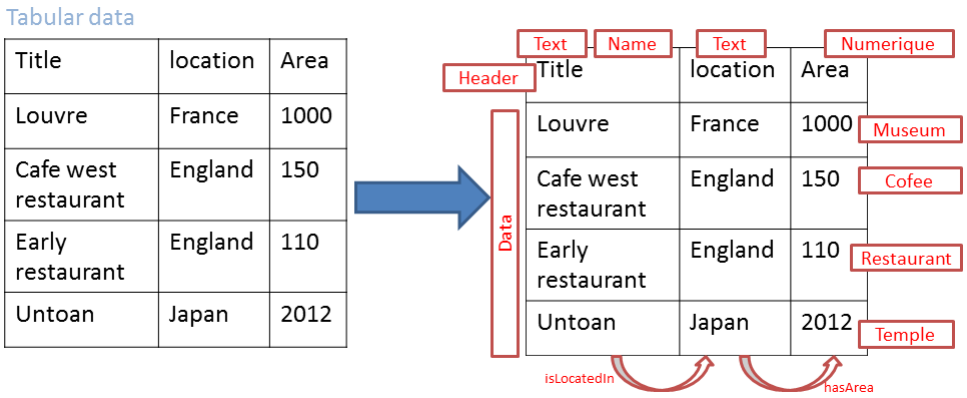


FIGURE 2.32 – Annotation d'une table Web.

Approche de Venetis, A. et al. (2011) : Retrouver la sémantique des tables sur le Web [48].

Cette approche enrichit une table Web avec des annotations supplémentaires telles que les noms de colonnes. Un libellé de type est donné à une colonne, si un nombre suffisant de valeurs de la colonne est identifié avec cette étiquette dans la base de données des étiquettes de types, qui est une source externe de référence. Par exemple, une colonne avec les valeurs suivantes : *Victor Hugo*, *Alexandre Dumas*, *Émile Zola*, sera annotée comme *Écrivain*. La méthode exige que les valeurs de colonne soient connues dans la source de référence afin d'attribuer une étiquette à la colonne. La méthode n'utilise pas les valeurs des autres colonnes de la ligne pour identifier le type d'une colonne. Pour des données RDF, le label d'une propriété est toujours présent, ce qui équivaut au nom de la colonne. Cependant, la valeur d'une propriété n'est pas toujours fournie pour toutes les instances d'un type. Quand les valeurs des propriétés sont présentes, elle peuvent

être exploitées par cette méthode pour trouver le label d'un cluster.

Approche de Limaye, G. et al. (2010) : Annotation des tables Web en utilisant ses instances, types et relations [49].

G. Limaye et al. [49] proposent une approche pour annoter les cellules (instances), colonnes (types) et relations binaires dans les tables Web, afin que la recherche dans ces tables soit plus facile. Les étiquettes sont prises de la base de connaissance *YAGO* [50], qui contient environ 250.000 types, deux millions d'instances et 99 relations. Une base de connaissance comprend les éléments suivant :

- L'ensemble des types T : les URIs d'un type sont mises sous forme canonique, car il peut en exister plusieurs variantes. Ces variantes sont représentées dans l'ensemble $L(T)$;
- L'ensemble des instances E : une instance peut avoir un ou plusieurs types. Chaque instance a également un ensemble associé à $L(E)$;
- L'ensemble des relations binaires B : chaque paire de colonnes de la table Web source doit être marquée avec des noms de relations canoniques. Le schéma de B s'écrit $B(T_i, T_j)$ avec T_i, $T_j \in T$. Un tuple de B est annoté : $B(E_i, E_j)$ avec E_i, $E_j \in E$, E_i de type T_i, et E_j de type T_j.

L'étiquetage de la table Web à partir de la base de connaissance se fait comme suit :

- Une cellule D_{rc}, de la ligne r et de la colonne c de la table Web, est étiquetée par une instance E_i de la base de connaissance en calculant la similarité entre le texte de la cellule D_{rc} et les instances de la base de connaissance ;
- L'en-tête d'une colonne c, noté H_c, est étiqueté en calculant la similarité entre le texte de H_c et les types T de la base de connaissance ;
- Une paire de colonnes C_i, C_j est étiquetée par une relation binaire $B(C_i, C_j)$, en cherchant dans la base de connaissance, les propriétés liant les deux types assignés aux deux colonnes. Par exemple, il existe $B(T_i, T_j)$ dans la base de connaissance avec C_i étiquetée avec T_i et C_j étiquetée avec T_j.

L'idée principale de ce travail est d'utiliser l'inférence commune sur chacun des composants individuels pour améliorer la qualité globale des étiquettes. Deux idées sous-tendent cette approche :

- Vérifier la compatibilité de l'assignation : (i) du type T_i avec l'en-tête H_c et (ii) de l'instance E_j avec la cellule D_{rc}. Cela par le fait d'avoir l'instance E_j de type T_i dans la base de connaissance ;
- L'existence de deux instances $E_i \in C_i$ et $E_j \in C_j$ ayant la relation $B(E_i, E_j)$ dans la base de connaissance va confirmer l'attribution de cette relation binaire aux deux colonnes C_i et C_j.

La méthode proposée n'annote les données qu'avec des étiquettes prises de *YAGO*. Cela limite donc son utilisation aux seuls cas où la table contient des données contenues dans *YAGO*.

Approche de Hignette, G. et al. (2009) : Annotation floue des tables Web en utilisant des ontologies [51].

Hignette, G. et al. [51] proposent également une approche pour annoter les tables Web afin de mieux les expliquer. Ceci à l'aide d'un système non supervisé qui utilise une ontologie pour attribuer des annotations floues à des tables Web (cellule, colonnes, relation entre colonnes). Une ontologie du même domaine que la table Web est considérée pour l'enrichir comme suit :

- Les synonymes des types sont ajoutés dans la liste des synonymes ;
- Pour les types numériques, une liste est ajoutée pour les unités et les intervalles si possible, ex : [0, 50] ;
- Les propriétés sont décrites par leur signature qui représente leur domaine/co-domaine.

La mesure de similarité cosinus est utilisée pour évaluer la similarité entre deux termes. Le problème est que chaque domaine a sa propre ontologie et qu'il n'existe pas une ontologie unique universelle utilisable. L'approche est dons inapplicable dans le cas où une ontologie adaptée à la table Web n'est pas trouvée.

Approche de Quercini, G. et al. (2013) : Annotation des instances dans les tables [52].

Quercini, G et al [52] proposent d'entraîner un algorithme d'apprentissage supervisé pour rechercher des informations sur des instances ambiguës sur le Web, c'est à dire dont le type n'est pas clairement identifié. Cela afin de les annoter avec le bon type. L'algorithme se compose de trois étapes principales :

- **Prétraitement :** les cellules dont le contenu n'est pas susceptible de représenter les noms des instances sont exclues de la recherche sur le Web, ex : date, URL, adresse, e-mail, etc. Ceci est fait en considérant l'expression régulière dans chaque cellule, et les types des colonnes (emplacement, date, nombre) dans lequel ils se produisent dans Google Fusion Tables (GFT) [53]. Ce dernier est un service récemment lancé par Google. Il représente une grande collection de tables avec des données provenant de différents domaines.
- **Annotation :** le contenu des cellules restantes est soumis au moteur de recherche Web *Microsoft Bing*. Le choix de ce moteur de recherche est dû au fait qu'il fournit une API qui impose moins de restrictions sur l'allocation des demandes quotidiennes que les autres moteurs de recherche. Le top-k des mots-clés extraits des pages retournées sont triés dans un classifieur de texte pour trouver les types décrits. Un type est assigné à une instance si $k/2$ des extraits le désignent. Le poids d'un type d'instance est calculé par le nombre d'extraits qui le désignent divisé par k (tous les extraits). Étant

donné un ensemble de type T, pour chaque $t \in T$, le classifieur est formé comme suit :
  - Un ensemble P, qui contient les instances de type t (instances positives), est créé. Les instances sont obtenues à partir de *DBpedia* ;
  - Jusqu'à 10 extraits sont collectés auprès de Bing pour chaque instance $e \in P$. La requête soumise à Bing est une expression obtenue par la concaténation du nom de e et le nom de t. Par exemple, si le nom de e est *Melissa* et du nom de t est *restaurant*, la demande à Bing est *MelissaRestaurant*. La raison est que le nom du type désambiguïse la demande afin de s'assurer d'obtenir des extraits qui décrivent l'instance visée.
- **Post-traitement :** les annotations erronées sont éliminées car les types de données dans les cellules d'une même colonne doivent être homogènes dans des tables bien formées. Les instances ayant des types avec un faible poids par rapport à d'autres sont considérés comme fausses.

La méthode proposée ici [52] complète les approches présentées précédemment [48, 49, 51]. En effet, ces approches annotent les instances dans une table en utilisant une base de connaissance de référence. Par conséquent, elles sont incapables de découvrir et d'annoter les instances qui n'appartiennent pas à la base de connaissance de référence. La méthode propose pour cela d'entraîner un algorithme pour rechercher des informations sur des instances inédites sur le Web de façon à les annoter avec le bon type. Cependant, la méthode trouve des types pour les instances, mais elle n'affecte pas plusieurs types à une instance et l'ensemble de types T est prédéfini car il est fixé au niveau de l'ensemble d'apprentissage du classifieur. La méthode se base sur le *nom* d'une instance pour trouver ses descripteurs sur Internet qui lui permettront de la classer à travers un classifieur formé sur un ensemble fini de types. Toutefois, pour les données qui n'ont pas de *nom*, par exemple, pour une instance de type *Présentation* décrite par (isRoleAt :tutorial2, heldBy :jon-Phipps), la méthode ne peut pas l'annoter ou l'annotera comme *Personne*. En plus, le *nom* de l'instance doit être connu sur le Web pour attribuer un type. Par exemple, soient les deux instances suivantes représentant des bâtiments :

- (nom : Fermat, zone : 200, nombre de salle : 14) ;
- (nom : Buffon, zone : 100, nombre de salle : 10).

Ces deux instances ne peuvent pas être annotées comme *Bâtiments de UVSQ* car ni *Fermat* ni *Buffon* ne donnent ce type sur le Web.

Dans d'autres cas, les annotations attribuées ne sont pas assez spécifiques, par exemple les types *secrétaire*, *enseignant* et *étudiant* seront étiquetés comme *personne*, alors que si l'on considère la structure des propriétés des instances de ces types, nous pouvons trouver trois labels différents.

2.5.3 Approches d'annotation de documents

Certains travaux sur l'annotation concernent les clusters de documents. L'idée principale de ces approches est d'annoter un groupe de documents similaires avec les mots les plus fréquents contenus dans le texte de ces documents. Dans [54, 55, 56], un cluster est annoté avec plusieurs termes pondérés contenus dans le document le plus représentatif du cluster. Un poids est attribué à chaque terme à l'aide d'une métrique dérivée de la F mesure, le rappel et la précision dans [54] et en utilisant tf-idf dans [55]. La précision d'un terme pour un cluster est le nombre de documents contenant le terme dans ce cluster. Le rappel d'un terme pour un cluster est le nombre de documents contenant ce terme dans d'autres clusters.

Approche de Carmel et al. (2009) : Annoter un cluster de documents en utilisant *Wikipedia* [57].

Dans [57], les annotations sont extraites de *Wikipedia* et des termes importants dans les documents. Étant donné un ensemble de documents, les termes les plus importants sont d'abord extraits du texte. Ensuite, les pages *Wikipedia* associées sont identifiées à l'aide d'une requête utilisant les termes importants dans les documents. Les catégories de *Wikipedia* et les titres de pages connexes servent de candidats potentiels pour l'annotation des clusters. En outre, tous les termes importants extraits du texte du cluster sont également considérés comme des candidats.

Approche de Treeratpituk, Pucktada et al. (2006) : Annotation de la hiérarchie de clusters de documents [58].

L'approche présentée dans [58] attribue des annotations à des clusters hiérarchiques de documents. Les mots ou expressions qui sont très fréquents dans le cluster, mais relativement rares dans la collection, sont sélectionnés comme annotations pour le cluster. En effet, une bonne annotation doit être relativement fréquente dans le cluster parent, mais très fréquente dans le cluster lui-même.

2.5.4 Approches d'annotation de texte

Plusieurs outils ont été proposés pour l'annotation d'une instance, comme *Wikipedia* Miner [59], *DBpedia* Spotlight [60] et *TAGME* 2 [61]. Ces outils annotent un texte en considérant une instance comme un mot dans un texte et l'annotation comme la définition de ce mot. La plupart de ces approches tentent d'identifier des mots importants d'un texte et elles les annotent par une définition à partir d'une source de connaissances telle que *Wikipedia*.

Comme exemple de ces outils d'annotation de texte, *TAGME* est un outil puissant capable d'identifier à la volée des phrases significatives courtes, appelées « spots », dans un texte non structuré et de les relier à une page *Wikipedia* pertinente d'une manière rapide et efficace. Ce processus d'annotation a des implications qui vont bien au-delà de l'enrichissement du texte avec des liens explicatifs car il concerne la contextualisation et, en quelque sorte, la compréhension du texte.

FIGURE 2.33 – Annotation de l'instance Diego Maradona avec l'outil *TAGME*.

La figure 2.33 représente un extrait de texte, sur le footballer *Diego Maradona*, tagué avec l'outil *TAGME*. L'outil *TAGME* identifie d'abord les mots importants du texte qui sont soulignés dans la figure. Puis, chaque mot est défini à partir de la source *Wikipedia*, comme le mot *Maradona* dans la figure.

2.5.5 Discussion sur les approches d'annotation

Plusieurs approches d'annotation visant à mieux expliquer les données, sont proposées dans des contextes différents. Une synthèse de ces approches est présentée dans le tableau 2.7.

Les approches d'annotation des données liées ne fournissent pas une méthode dédiée à l'annotation des instances mais plutôt une façon de trouver le label d'un groupe découvert par une approche d'extraction de schéma. Dans [2], les groupes de catégories découverts sont annotés par les déclarations *rdf*:*type* et les déclarations *dcterm*:*subject*. Si une déclaration est la même pour toutes les instances d'un cluster, alors elle est considérée comme un label du cluster. Cependant, ces déclarations sont souvent incomplètes et voire même inexistantes. Dans [31], les arcs entrants les plus fréquents pour les données OEM [25] sont considérés, ce qui équivaut à considérer les déclarations les plus courantes de *rdf*:*type* dans un jeu de données RDF, comme proposé dans [26]. Cependant, ces déclarations sont souvent manquantes ou incomplètes.

Les approches proposées dans le contexte des tables Web, peuvent être adaptées aux données RDF. Cela en considérant une ligne comme une instance, les

Contexte	Idée générale	Conclusion
– Données Liées	– Déclarations *rdf:type* et *dcterm:subject* identiques dans un groupe [2] – Déclaration *rdf:type* la plus fréquente dans le groupe [26] – Le label des arcs entrants [31]	– Limité par les informations contenues dans le jeu de données
– Tables Web	– Bases de connaissance externes : ontologie dans le domaine [51], *YAGO* [49], Bing [52] – Exploitation des entêtes des colonnes [51, 49] – Exploitation des valeurs des données [48] – Construction d'un classifieur sur la base de recherche Web sur le Nom des instances [52]	– Les tables Web ont une structure homogène – Limité par un nombre fini de types dans un classifieur [52]
– Clusters de documents	– Extraire les mots les plus fréquents dans le documents du cluster	– Les annotations qui décrivent le mieux les données RDF ne sont pas nécessairement présentes dans la source de données, ni dans les propriétés ni dans les valeurs
– Texte	– Une instance est un mot dans un texte et l'annotation est la définition de ce mot – Utilisation de *Wikipedia* comme base de connaissance [60, 61]	– L'annotation que nous entendons n'est pas la définition d'un mot

TABLE 2.7 – Synthèse des approches d'annotation présentées

titres de colonne comme des propriétés et les valeurs de colonnes comme des valeurs de propriétés. L'approche proposée dans [48] utilise uniquement les valeurs des données, et celles proposées dans [49, 51] prennent en compte l'en-tête des colonnes et les relations possibles entre les colonnes. Dans [52], un algorithme d'apprentissage est utilisé pour rechercher des informations sur des instances inhabituelles sur le Web, en fonction de leurs noms, afin de les annoter avec le bon type. Cependant, cette approche est limitée par un classifieur qui est formé sur un ensemble fini de types. De plus, contrairement aux sources de données RDF, les données tabulaires sont structurées. Ce qui est intéressant dans ces approches est qu'elles extraient les étiquettes à partir de sources de connaissances externes, ce qui n'existe pas pour les données RDF.

L'idée principale des approches d'annotation de documents est d'annoter un groupe de documents similaires avec les mots les plus fréquents contenus dans le texte de ces documents. Cependant, les annotations qui décrivent le mieux les données RDF ne sont pas nécessairement présentes dans l'ensemble de données, ni dans les propriétés ni dans les valeurs.

Les outils proposés pour l'annotation de texte considèrent une instance comme un mot dans un texte et l'annotation comme la définition de ce mot. Cependant, nous entendons par une annotation, un label qui décrit une instance ou un groupe d'instances en capturant leur sens. Ce qui pourrait compléter les approches existantes pour la découverte du schéma implicite comme [26, 2, 28, 30, 3, 31, 32, 4, 5, 34], par une méthode qui permet de fournir des noms pour les groupes découverts.

2.6 Conclusion

Nous avons présenté dans cet état de l'art un ensemble d'approches relatives à l'extraction de schéma. Nous nous sommes intéressés aux (i) approches de découverte du schéma implicite ; (ii) aux approches d'enrichissement du schéma explicite ; (iii) aux approches de découverte des patterns structurels des instances d'un jeu de données ; et enfin (vi) aux approches qui permettent de produire des annotations dans différents contextes dans le but d'expliciter la sémantique d'un groupe découvert ou d'une structure.

2.6.1 Bilan sur les approches existantes

Les approches qui découvrent le schéma d'un ensemble de données à partir de la structure implicite des données ne requièrent aucune information sur le schéma du jeu de données. Certaines de ces approches [26, 2, 28, 30, 3, 31] procèdent par le regroupement des instances similaires, tandis que d'autres [32, 4, 5, 34] procèdent par le regroupement des chemins dans le graphe de données. La plus part des approches qui procèdent par regroupement des instances similaires structurellement

requièrent des paramètres de regroupement comme un seuil de coupure [26], un seuil de similarité [3], ou le nombre de cluster voulu [31]. Les approches [28, 30] utilisent FCA et par conséquent aucun paramètre n'est à définir. Cependant, avec FCA les concepts découverts ne reflètent pas forcément les types contenus dans le jeu de données, et les liens entre les concepts du treillis construit ne reflètent pas les liens entre les types. Le treillis découvert avec FCA peut être très grand et il représente plus un index sur les données qu'un schéma compact des données. L'approche proposée dans [2] ne requiert pas de paramètre mais elle découvre plutôt des catégories et elle les ordonne sous forme hiérarchique. Cependant, si on voulait par exemple connaître les catégories de base du jeu de données, un seuil de coupure sur l'arbre hiérarchique serait requis comme pour [26].

Certaines approches [32, 4, 34, 5] découvrent le schéma implicite d'un ensemble de données en regroupant les chemins similaires d'un graphe de données. Certaines de ces approches construisent un dataguide [32] ou un dataguide approximatif [4]. Le problème du dataguide est qu'il peut être beaucoup plus grand que le graphe de données initial. Le problème du dataguide approximatif est qu'il est construit en utilisant COBWEB qui est un algorithme non déterministe, et que des instances de types différents peuvent être regroupées. L'approche [34] utilise la bisimulation pour réduire la taille d'un graphe. Des instances de même type ne sont pas forcément regroupées car il faut qu'elles aient exactement les même arcs sortants et leur enfants également. L'approche [5] est fondée sur la généralisation des chemins avant de les regrouper. Cependant, cette approche n'est pas totalement automatique. De manière générale, les approches qui regroupent les chemins similaires d'un graphe de données manquent de précision par rapport au schéma extrait en terme de types découverts et de liens entre eux : les liens sont généralement redondants et les groupes découverts sont soient très homogènes et redondants, soient très hétérogènes. En plus, il n'y pas de paramètre pour réguler l'hétérogénéité des groupes découverts.

Les approches [26, 2, 28, 30, 3, 31] qui découvrent le schéma implicite d'un ensemble de données en regroupant les instances similaires structurellement sont plus adaptées au contexte de la découverte du schéma d'un graphe de données comme RDF. Alors que les approches [32, 4, 34, 5] qui regroupent les chemins en commun sont plus adaptées aux données présentées sous forme arborescente comme XML, où le schéma est plus sous la forme d'un dataguide. En effet, un schéma d'un graphe de données RDF est d'abord vu comme l'ensemble des types contenus dans le jeu de données avec des liens entre ces types.

Les approches [36, 40, 44, 38, 41, 43] qui enrichissent le schéma explicite d'un jeu de données requièrent des déclarations explicites sur le schéma comme $rdf{:}type$, $rdfs{:}range$, $rdfs{:}domain$, $rdfs{:}subClassOf$, etc. Le problème de ces approches est que ces déclarations sont souvent manquantes ou inexistantes dans un jeu de données RDF, et donc, elles ne peuvent pas être appliquées dans ces cas. Tandis que les approches qui découvrent le schéma implicite d'un jeu de données

le font à travers la structure inhérente des instances. Rappelons tout de même que la principale limite des approches [26, 3, 31] qui découvrent le schéma en terme de types et de liens entre eux est la spécification des paramètres et qu'une instance n'est assignée qu'à un seul type, alors qu'une instance dans une source RDF peut être de plusieurs types.

Nous avons présenté un ensemble d'approches de découverte de patterns structurels exacts [6, 7, 8] et un ensemble d'approches de découverte de patterns structurels approximatifs [45, 46, 10] à partir d'un jeu de données. La différence majeure entre les deux familles est que dans un pattern approximatif, la co-occurrence des propriétés n'est pas renseignée. En effet, pour les approches de découverte de patterns approximatifs, celle présentée dans [45] marque les propriétés optionnelles d'un pattern, tandis que l'approche proposée dans [10] ne les marque pas, et celle proposée dans [46], n'inclut pas les propriétés optionnelles dans la description du pattern. L'approche de découverte de patterns structurels exacts présentée dans [6] traite les sources de données locales, celle présentée dans [7] traite les données en flux, et celle présentée dans [8] concerne des sources de données distribuées. Toutes ces approches ont accès aux données et doivent les parcourir. Elles sont inapplicables sur une source de données distante où l'accès aux données est restreint en terme de temps d'exécution et de nombre de requêtes envoyées.

Nous avons présenté un ensemble d'approches d'annotation selon différents contextes. Les approches qui concernent les données liées [2, 31, 26] ne fournissent pas une méthode dédiée à l'annotation des instances mais plutôt une façon de trouver le label d'un groupe découvert par une approche d'extraction de schéma. Les approches proposées dans le contexte des tables Web [48, 49, 51, 52], pourraient être adaptées aux données RDF, mais contrairement aux jeux de données RDF, les données tabulaires sont structurées. L'idée principale des approches d'annotation de documents [54, 55, 56, 57, 58] est d'annoter un groupe de documents similaires avec les mots les plus fréquents contenus dans le texte de ces documents. Cependant, les annotations qui décrivent le mieux les données RDF ne sont pas nécessairement présentes dans l'ensemble de données, ni dans les propriétés ni dans les valeurs.

2.6.2 Problèmes ouverts

A travers notre étude de l'état de l'art sur la découverte d'information sur le schéma d'un jeu de données liées, nous avons identifié plusieurs limites que nous discutons dans ce qui suit.

Les approches d'enrichissement de schéma requièrent des déclarations explicites sur le schéma dans le jeu de données pour pouvoir les enrichir et les compléter. Cependant, ces déclarations sont souvent manquantes et voir inexistantes. Les approches qui découvrent le schéma implicite d'un jeu de données liées ne re-

quièrent aucune déclaration sur le schéma. Cependant, ces approches présentent principalement les limites suivantes : (i) un paramètre de regroupement est à spécifier ; (ii) un seul type est attribué à une instance ; (iii) elles ne sont pas incrémentales. Un problème qui reste ouvert est la caractérisation de la structure implicite d'une source de données sans faire aucune hypothèse sur l'existence de déclarations sur le schéma dans la source de données. De plus, il faudrait que l'approche soit peu coûteuse et adaptée aux données du Web sémantique où les instances d'un même type sont décrites par des ensembles de propriétés hétérogènes. Les sources de données du Web sémantique sont généralement en constante évolution, le schéma implicite découvert doit être capable de suivre l'évolution des instances. L'approche ne doit pas nécessiter de paramètres, car pour être capable de spécifier des paramètres il faut avoir des connaissances sur le jeu de données, ce qui n'est pas évident pour des données du Web sémantique. L'approche doit être également capable d'attribuer plusieurs types à une instance.

Les instances du même type peuvent être décrites par des ensembles de propriétés différents. La caractérisation d'un type dans un schéma par les propriétés qui décrivent ses instances ne suffit pas pour refléter la co-occurrence entre ses propriétés. En effet, si par exemple dans la source de données des personnes sont caractérisées par un nom, une adresse et un mail ; rien ne garantit que dans la source de données, il existe des personnes décrites par une adresse et un mail en même temps. Le renseignement de la co-occurrence entre les propriétés d'un type est importante pour, par exemple, savoir si une source données contient bien les informations recherchées, et cela avant de l'interroger. Les approches de découverte de patterns structurels ont été proposées pour des sources de données locales, des sources de données distribuées et des données en flux. Toutes ces approches nécessitent de parcourir les données pour découvrir les patterns. Ces approches ne sont pas applicables en ligne sur une source de données distante avec un accès restreint aux données à travers des requêtes uniquement. La difficulté de cette tâche réside dans les restrictions d'accès par le serveur comme des timeout sur l'exécution d'une requête et la limitation du nombre de requêtes envoyées pour ne pas engorger le réseaux.

L'annotation du schéma découvert est importante pour pouvoir le comprendre. Les approches d'annotation proposées présentent les limites suivantes pour l'annotation d'un schéma : (i) les approches d'annotation pour les données liées utilisent des déclarations sur le schéma qui ne sont pas toujours disponibles ; (ii) les approches d'annotation pour les tables Web sont intéressantes, cependant, ces données sont homogènes contrairement aux données liées ; et (iii) les approches d'annotation pour les documents extraient des labels à partir du texte des documents, cependant, un label pour des données liées n'est pas dans leurs propriétés ni dans leurs valeurs. Par conséquent, il n'existe pas d'approche d'annotation dédiée aux données RDF et qui soit capable d'introduire de nouveaux types qui ne figurent pas dans la source de données. L'annotation pourrait être utilisée pour

annoter le schéma découvert d'un jeu de données liées. En effet, la structure seule ne suffit pas à comprendre tout le contenu d'une source de données. Certaines déclarations de types dans la source de données peuvent ne pas capturer par leur terminologie tout le sens des instances d'un type. Il est intéressant même dans ce cas où la déclaration est présente de l'expliquer davantage à partir de connaissances disponibles sur le Web.

Un autre problème ouvert est l'évaluation de la distance entre un schéma et la source de données qu'il décrit. En effet, les instances n'ont pas à suivre les déclarations sur le schéma si elles sont fournies dans la source de sonnées : une instance peut être décrite par des propriétés qui ne sont pas déclarées pour son type ; et elle peut élégamment ne pas être décrite par toutes les propriétés déclarées pour son type. Si le schéma d'une source de données est fourni, il est important de pouvoir caractériser l'écart existant entre le schéma et les données avant d'exploiter le schéma.

Chapitre 3

Découverte de schéma dans des sources de données RDF

3.1 Introduction

Le Web a évolué d'un ensemble de documents connectés par des liens hypertextes vers des données interconnectées, donnant naissance au Web des données. Un volume sans précédent de données est ainsi disponible, permettant la conception de nouvelles applications dans de nombreux domaines. Ces sources de données sont publiées suivant un ensemble de principes de bonne pratique [62] tels que : (i) l'utilisation des URIs (Uniform Resource Identifier) comme identifiant pour les ressources ; (ii) la possibilité de consulter ces URIs sur le Web et (iii) l'insertion de liens vers d'autres URIs. Des langages spécifiques ont été proposés par le W3C pour décrire les données du Web. RDF [1] (Resource Description Framework) est le langage proposé pour les données du Web sémantique. C'est un modèle de graphe destiné à décrire les ressources Web et leurs métadonnées. Avec le langage RDF, les données sont décrites sous-forme de triplets $s\ p\ o$, représentant respectivement un sujet, une propriété et un objet. Ceci peut être représenté par un graphe comme dans la figure 3.1, par exemple le triplet $P_3\ affiliation$ UVSQ, décrit le fait que la personne P_3 est liée à $UVSQ$ par la propriété $affiliation$. RDFS [2] ou RDF Schema (Resource Description Framework Schema) est une extension de RDF qui permet de structurer des ressources RDF. Chaque propriété peut être associée à un domaine et un co-domaine. Le domaine d'une propriété est spécifié avec une déclaration $rdfs{:}domain$, qui définit le type des sujets liés à une propriété. Le co-domaine d'une propriété est spécifié avec une déclaration $rdfs{:}range$, qui définit le type des objets liés à une propriété. OWL [3] (Web Ontology Language) est un langage qui permet de définir des ontologies Web. Une ontologie est la conceptualisation de la connaissance dans un domaine spécifique en représentant les concepts ainsi que les relations entre ces concepts.

1. Resource Description Framework : http ://www.w3.org/RDF/
2. RDF Schema : http : /www.w3.org/TR/rdf-schema/
3. Web Ontology Language : http ://www.w3.org/OWL/

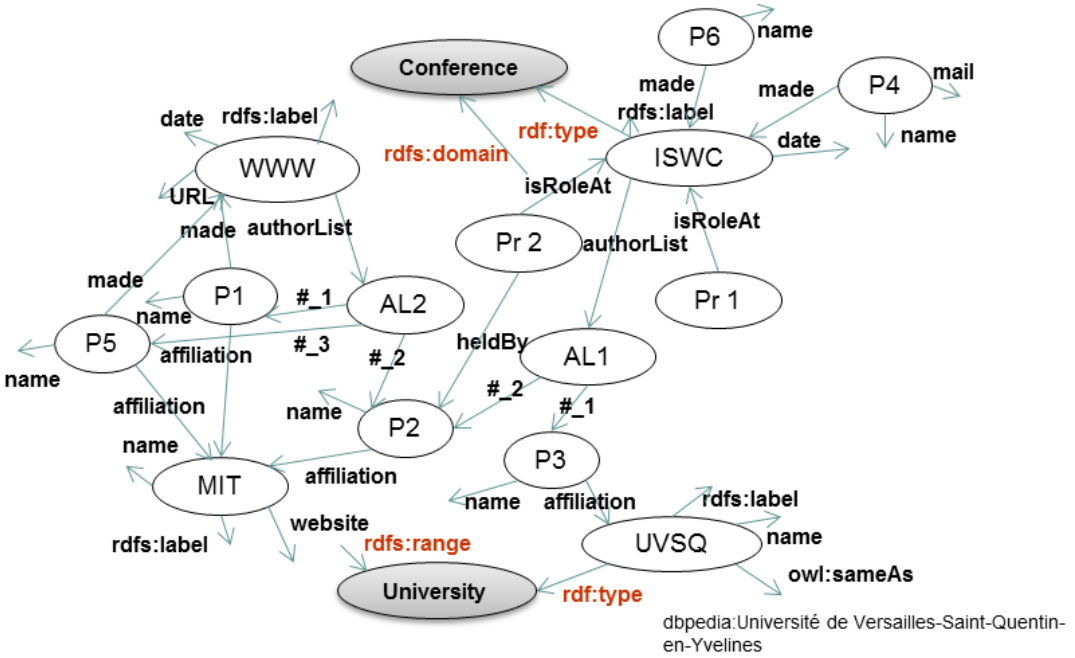


FIGURE 3.1 – Exemple d'un jeu de données liées.

Une source de données décrite en RDF, RDFS, et OWL contient à la fois les données et des informations sur le schéma. En effet, les déclarations sur le schéma sont décrites dans le jeu de données par des propriétés telles que : *rdf*:*type*, *rdfs*:*range*, *rdfs*:*domain*, *rdfs*:*subClassOf*, etc. La figure 3.1 montre un exemple d'un jeu de données liées. Les nœuds sont connectés en utilisant des liens représentant des propriétés. Un nœud peut représenter des types, comme *Conference* et *University*, ou des instances de types comme *UVSQ* ou *P1*. Certaines instances ont une déclaration *rdf*:*type* qui exprime leur appartenance à un type, comme l'instance *UVSQ* qui est liée au type *University*. Un jeu de données peut inclure un lien *owl*:*sameAs* qui permet de lier une URI du jeu de données à une URI dans un autre jeu de données, par exemple, l'instance *UVSQ* du jeu de données de la figure 3.1, est liée à une autre instance qui représente le même objet réel dans la source de données *DBpedia*. D'autres informations sur le schéma sont fournies dans cette source de données comme la déclaration *rdfs*:*range* pour la propriété *website*, qui la lie au type *University*. Cependant, ces déclarations sur le schéma sont souvent manquantes ou incomplètes, comme c'est le cas pour les instances *P3* ou *WWW*.

Le schéma d'une source de données est essentiel pour son exploitation. Un schéma permet de représenter le contenu d'une source de données en termes de types et de liens entre eux. Par exemple, pour pouvoir interroger la source de données de la figure 3.1, il est utile de connaître les types existants et leurs propriétés. Supposons qu'on veuille extraire les personnes qui travaillent à *MIT*

et qui ont fait une présentation à la conférence *ISWC*. La requête est formulée à l'aide du langage d'interrogation SPARQL comme suit :

- *<Select ?p where { ?p affiliation "MIT" . ?r heldBy ?p . ?r isRoleAt "ISWC"}>*.

La formulation de cette requête est impossible sans savoir : qu'une personne est liée à une université par la propriété *affiliation*, qu'une présentation est liée à une personne par la propriété *heldBy*, et qu'une représentation est liée à une conférence par la propriété *isRoleAt*. Plus généralement, la formulation d'une requête est impossible sans connaître le contenu d'une source de données en termes de types et de liens entre eux. Le schéma est également essentiel pour l'exécution de requêtes distribuées. Lorsqu'une requête est exécutée sur plusieurs sources de données, la disponibilité du schéma de chaque source caractérisant son contenu permet de décomposer la requête et d'envoyer les sous-requêtes aux sources pertinentes [21]. Le schéma est utile à d'autres fins, comme la création de liens entre les sources de données. En effet, il est recommandé lors de la publication d'une source de données sur le Web, d'y inclure des liens avec les données d'autres sources. Cela fait partie des principes des données liées afin de maximiser la navigabilité entre les sources et transformer le Web en un espace d'information unique et global. Il existe plusieurs outils d'interconnexion comme Knofuss [4] ou Silk [5], qui a été utilisé pour lier *Yago* [50] à *DBpedia* [63]. Cependant, ces outils se basent sur les déclarations de types existantes dans les sources pour générer des liens *owl*:*sameAs* appropriés. Si ces déclarations de types sont manquantes ou incomplètes dans la source de données, les résultats fournis seront incomplets.

Une source de donnée RDF(S)/OWL peut contenir des informations sur son schéma, comme nous l'avons vu dans la figure 3.1, à travers des déclarations. Cependant, ces informations ne sont pas toujours complètes. Même lorsque les jeux de données sont extraits de façon automatique, comme *DBpedia* qui est extrait automatiquement à partir des pages de *Wikipédia*, l'information sur le type peut être manquante. Les expérimentations présentées dans [40] montrent que tout au plus 63.7 % des données de *DBpedia* ont une information complète sur le type des données, et au plus 53.3 % dans YAGO. Ceci montre l'utilité de disposer d'une approche automatique pour trouver ces types.

Dans ce chapitre, nous adressons la problématique de la découverte automatique du schéma d'une source de données. Cette problématique a été abordée par plusieurs travaux présentés dans l'état de l'art. Cependant, les approches proposées présentent essentiellement deux limites. La première limite concerne les hypothèses faites par les approches qui infèrent ou enrichissent un schéma, et notamment l'existence de certaines déclarations sur le schéma dans la source de données. Ces approches infèrent de nouvelles déclarations sur le schéma à partir des déclarations déjà fournies dans la source de données. Cependant, ces déclarations

4. Knofuss : technologies.kmi.open.ac.uk/knofuss
5. Silk : wifo5-03.informatik.uni-mannheim.de/bizer/silk

sont souvent manquantes. La deuxième limite concerne la façon dont s'effectue le regroupement par les approches qui découvrent le schéma en regroupant les entités qui sont structurellement similaires. Ces approches n'affectent qu'un seul type à une entité, alors qu'en réalité, une entité RDF peut avoir plusieurs types. De plus, un algorithme de regroupement nécessite généralement la spécification de paramètres tels que le nombre de clusters pour l'algorithme k-means, ou le seuil de similarité pour l'algorithme de clustering hiérarchique.

Le but de notre travail est d'extraire le schéma décrivant une source de données RDF(S)/OWL. Nous entendons par schéma un descriptif des types et des liens entre eux contenus dans la source de données. Cette tâche n'est pas triviale pour plusieurs raisons. Dans le contexte des données du Web, la notion de schéma est vue comme un guide et non pas comme une représentation à laquelle les données doivent strictement se conformer. En effet, la nature des langages utilisés pour décrire les données du Web, tels que RDF(S)/OWL, n'imposent pas que les données les respectent. Par conséquent, les déclarations sur le schéma fournies dans une source de données peuvent ne pas refléter la structure réelle des entités de la source. De plus, ces déclarations sur le schéma sont souvent manquantes ou incomplètes, par exemple : rien ne garantit que les déclarations de type fournies dans la source de données soient assez riches pour permettre de trouver le type des données qui n'ont pas la déclaration du type. Pour cela, l'enrichissement à partir des déclarations de type existantes ne permet pas toujours d'inférer un descriptif structurel complet des données.

Afin de découvrir le schéma d'une source de données RDF(S)/OWL, nous avons opté pour une approche qui découvre la structure implicite des données. Cela en regroupant les données selon leur similarité structurelle pour découvrir les types contenus dans la source de données. Cette approche soulève plusieurs difficultés. D'abord, les données sont irrégulières, en effet, deux entités ayant le même type peuvent avoir des propriétés différentes. Ensuite, une entité peut avoir plusieurs types, mais, on ne connaît pas le nombre de types qu'elle peut avoir. Les sources de données du Web sont généralement ouvertes, par conséquent elles sont sujettes à une évolution constante. Enfin, ces sources de données peuvent contenir du bruit, et les données bruitées ne doivent pas altérer le schéma découvert. Pour toutes ces raisons, il est essentiel d'identifier la bonne méthode de regroupement qui va permettre de découvrir le schéma d'une source de données. Une approche de regroupement nécessite un critère de regroupement qui peut être le nombre de groupes voulus ou le seuil de similarité. Ces paramètres ne sont pas évidents à fixer, ce qui représente un verrou supplémentaire.

En plus des types contenus dans la source de données, un schéma doit également décrire les liens entre ces types, qui peuvent être de deux types : les liens sémantiques qui décrivent le domaine et le co-domaine d'une propriété, et les liens hiérarchiques qui décrivent la relation de généralisation entre deux types, comme entre les types *Auteur* et *Personne*. Cependant, le regroupement des données

selon leur similarité structurelle ne permet pas de générer ces liens.

Le chapitre est organisé comme suit. Nous introduisons une définition de notre problème dans la section 5.2. Nous présentons une analyse des algorithmes de clustering pour le regroupement des données liées dans la section 3.3. Notre approche générale de découverte automatique du schéma d'une source de données liées est présentée dans la section 3.4. Nous décrivons dans la section 3.5, les algorithmes de base de notre approche de découverte de types. La section 3.6 est consacrée à la génération des liens sémantiques et hiérarchiques pour compléter le schéma découvert. Nos résultats d'évaluation sont présentés dans la section 5.6. Enfin, nous présentons le bilan de notre approche de découverte de schéma et nous concluons ce chapitre dans la section 6.10.

3.2 Formalisation de la problématique

La problématique abordée dans ce chapitre peut être énoncée comme suit : étant donné une source de données liées décrite en RDF(S)/OWL avec des informations incomplètes sur le schéma, comment peut-on découvrir les types des données et les liens entre eux ?

Nous présentons d'abord les caractéristiques d'une source de données liées dans la section 3.2.1. Puis, nous présentons les caractéristiques du schéma à découvrir dans la section 3.2.2. Enfin, les caractéristiques du processus de découverte de schéma ainsi que les défis posés sont présentés dans la section 3.2.3.

3.2.1 Description d'une source de données

Dans une source de données liées décrite en RDF(S)/OWL, on trouve à la fois les données et les informations sur le schéma lorsqu'elles sont fournies. En effet, la source peut contenir des déclarations sur le schéma telles que : rdf:*type*, $rdfs$:*range*, $rdfs$:*domain*, $rdfs$:$subClassOf$, etc. Nous définissons une source de données liées comme suit.

Définition (Source de données liées). Une source de données liées est représentée par un graphe orienté et étiqueté G, où chaque nœud est une ressource, un nœud blanc (anonyme, vide) ou un littéral. Considérons les ensembles R, B, P et L représentant des ressources, des nœuds blancs, des propriétés et des littéraux respectivement. Un jeu de données décrit en RDF(S)/OWL est défini comme un ensemble de triplets $D \subseteq (R \cup B) \times P \times (R \cup B \cup L)$. Chaque arc dans le graphe, reliant un nœud e_1 à un autre nœud e_2 étiqueté par la propriété p représente le triplet (e_1, p, e_2) de la source de données D. Dans ce graphe, nous définissons une entité comme un nœud correspondant à une ressource.

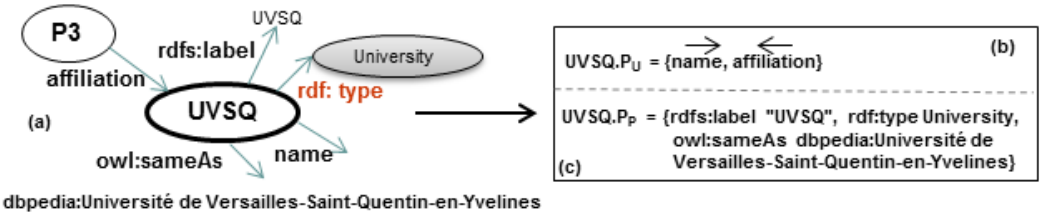


FIGURE 3.2 – Exemple de description d'une entité.

La figure 3.1 montre un exemple d'une telle source de données, décrivant des conférences. Nous pouvons voir que certaines entités sont décrites par la propriété *rdf:type*, définissant les types auxquels elles appartiennent, comme c'est le cas pour l'entité *UVSQ*, qui est définie comme une université, pour d'autres entités, telles que *WWW*, cette information sur le type de l'entité est manquante. Deux entités ayant le même type ne sont pas nécessairement décrites par les mêmes propriétés, comme nous pouvons le voir pour les entités *UVSQ* et *MIT* dans notre exemple, qui sont toutes deux des universités, mais contrairement à l'entité *UVSQ*, l'entité *MIT* est décrite par la propriété *website*.

Nous considérons qu'une entité est décrite par des propriétés de nature différente. Certaines d'entre elles font partie des vocabulaires RDF(S)/OWL, et nous les appelons propriétés primitives, comme la propriété *rdfs:label*. Les autres propriétés sont définies par l'utilisateur, comme la propriété *affiliation* de la figure 3.1. Nous faisons la distinction entre ces deux types de propriétés car toutes les propriétés ne doivent pas être utilisées de la même façon au cours du processus de découverte de types. Les propriétés primitives comme *rdfs:label* peuvent être définies pour n'importe quelle instance. Cependant, une propriété utilisateur, comme la propriété *affiliation* ne décrit que des personnes. Si deux entités sont décrites par les mêmes propriétés primitives, on ne peut pas en conclure qu'elles sont de même type. Par conséquent, ces propriétés ne doivent pas être considérées lors de l'évaluation de la similarité entre les entités. Certaines propriétés primitives peuvent fournir des informations sur le schéma de la source de données, comme *rdf:type*, *rdfs:range*, *rdfs:domain*, *owl:sameAs*, etc. Si elles sont présentes dans la source de données, elles peuvent être utilisées pour valider le schéma résultant, par exemple si deux entités e et e' sont liées avec la propriété primitive *owl:sameAs*, alors nous pouvons vérifier que les types inférés pour e et e' sont les mêmes. Nous décrivons formellement une entité comme suit.

Définition (Description d'une entité). Étant donné l'ensemble des propriétés primitives P_P et l'ensemble des propriétés définies par l'utilisateur P_U dans la source de données D, une entité e est décrite par :

1. L'ensemble des propriétés définies par l'utilisateur $e.P_U$ composé de pro-

priétés $p_u \in P_U$, chacune annotée par une flèche indiquant sa direction, tel que :
- Si $\exists(e, p_u, e') \in D$ alors $\overrightarrow{p_u} \in e.P_U$;
- Si $\exists(e', p_u, e) \in D$ alors $\overleftarrow{p_u} \in e.P_U$.

2. L'ensemble des propriétés primitives $e.P_P$, composé de propriétés $p_p \in P_P$ et de leurs valeurs, tel que :
- Si $\exists(e, p_p, e') \in D$ alors $p_p \in e.P_P$.

Cette notion d'entité est utilisée dans le reste du manuscrit pour designer une instance d'un type.

La figure 3.2 montre un exemple de description d'une entité. Dans le but de découvrir les types des données, les entités sont comparées selon leur structure. Notre algorithme de découverte de types prend en entrée l'ensemble $D_U = \{e_i.P_U : i = 1, ...n\}$, où $e_i.P_U$ représente l'ensemble des propriétés définies par l'utilisateur décrivant l'entité e_i. L'ensemble $D_P = \{e_i.P_P : i = 1, ...n\}$, où $e_i.P_P$ représente l'ensemble des propriétés primitives décrivant l'entité e_i. Il peut être utilisé pour valider les résultats du schéma inféré à travers des règles. En effet, une fois le schéma découvert, une phase de validation permet de vérifier que le schéma découvert ne contredit pas les déclarations sur le schéma fournies dans la source de données ; par exemple deux entités ayant la même déclaration de type dans la source de données, doivent appartenir à un même type dans le schéma découvert. Ce processus de validation n'est pas abordé dans ce chapitre.

3.2.2 Description d'un schéma

Un schéma permet de résumer le contenu d'une source de données en termes de types et de liens entre eux. Nous définissons un schéma comme suit.

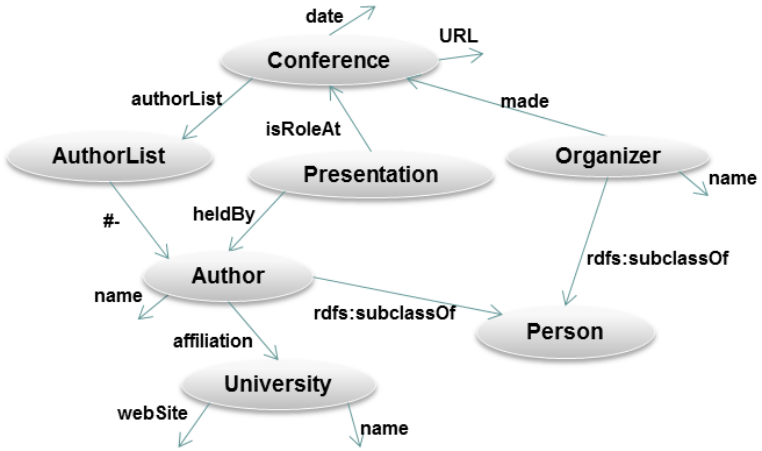


FIGURE 3.3 – Le schéma découvert pour la source de données de la figure 3.1.

Définition (Schéma). Le schéma extrait S pour une source de données D est composé de :

- Un ensemble de types qui se recouvrent éventuellement $T = \{T_1, ..., T_n\}$, où chaque types T_i correspond à un ensemble d'entités de D de même type ;
- Un ensemble de liens $\{p_1, ..., p_m\}$, où chaque p_i est une propriété pour laquelle le co-domaine et le domaine correspondent à deux types dans T ;
- Un ensemble de liens hiérarchiques, où chaque lien hiérarchique concerne deux types dans T, exprimant que l'un est le type générique de l'autre.

TABLE 3.1 – Les types des entités du jeu de données *Conference* (Figure 3.1).

Types	Entités
Conference	*WWW*, *ISWC*
Presentation	*Pr1*, *Pr2*
AuthorList	*AL1*, *AL2*
University	*UVSQ*, *MIT*
Organizer	*P1*, *P4*, *P5*, *P6*
Author	*P1*, *P2*, *P3*, *P5*
Person	*P1*, *P2*, *P3*, *P4*, *P5*, *P6*

La figure 3.3 représente le schéma découvert pour le jeu de données de la figure 3.1 qui comporte sept définitions de types. Les ensembles respectifs des entités correspondants à chaque type sont donnés dans le tableau 3.1. Comme nous pouvons le voir dans les résultats, les types ne sont pas nécessairement disjoints : l'entité *P1* a trois types, à savoir *Author*, *Organizer* et *Person*. La découverte du schéma d'un jeu de données demande également l'identification des liens entre les types. Dans notre exemple, comme les entités *P4* et *ISWC* sont liées par la propriété *made*, nous pouvons en déduire l'existence d'un lien sémantique étiqueté *made* entre les types *Organizer* et *Conference* dans le schéma résultant. Un autre type de lien possible est le lien hiérarchique. En effet, les types peuvent être liés dans le schéma extrait par une propriété $rdfs{:}subClassOf$, comme entre le type générique *Person* et ses types spécifiques *Author* et *Organizer* de la figure 3.3.

3.2.3 Processus de découverte de schéma et défis

Nous adressons la problématique de la découverte d'information sur le schéma d'une source de données liées. Après l'étude de l'état de l'art, nous avons identifié deux façons de procéder. La première consiste à utiliser certaines déclarations sur le schéma existantes dans la source de données, comme dans [40, 36, 43, 8]. Ces approches infèrent ou enrichissent un schéma en utilisant des déclarations existantes dans la source de données. Le problème dans ces méthodes est que ces

déclarations sont souvent manquantes, ou incomplètes, comme pour certaines instances de la figure 3.1. La deuxième façon de procéder est de découvrir le schéma à partir de la structure implicite des données sans utiliser de déclarations sur le schéma, comme décrit par les approches [31, 26, 4]. Ces approches présentées dans l'état de l'art découvrent le schéma en regroupant les entités qui sont similaires structurellement. Les limites de ces approches sont qu'elles n'affectent qu'un seul type à une entité, alors qu'en réalité, une entité dans une source RDF peut avoir plusieurs types. De plus, certains algorithmes de regroupement utilisent des paramètres difficiles à fixer tels que le nombre de clusters pour l'algorithme k-means dans l'approche proposée dans [31] ; ou le seuil de similarité pour l'algorithme hiérarchique dans l'approche proposée dans [26].

Comme les informations sur le schéma dans une source de données liées sont souvent manquantes ou incomplètes, nous proposons de découvrir le schéma à partir de la structure implicite des données sans utiliser de déclarations sur le schéma dans la source de données. Cela pose plusieurs défis liés à l'irrégularité des données : deux entités ayant le même type peuvent avoir des propriétés différentes et une entité peut avoir plusieurs types. De plus, les sources de données du Web sont sujettes à une évolution constante et peuvent contenir du bruit.

Dans notre approche, la découverte de schéma est fondée sur le regroupement des entités selon leur similarité structurelle. Nous proposons d'appliquer un algorithme de clustering pour former ces groupes d'entités. Cependant, un algorithme de clustering nécessite un critère de regroupement qui peut être le nombre de groupes voulus ou le seuil de similarité pour le regroupement. Comme on ne connaît pas *a priori* le nombre de types contenus dans la source de données, ces paramètres ne sont pas évidents à fixer, et cela représente en soi un défi supplémentaire. Pour toutes ces raisons, il est essentiel d'identifier la bonne méthode de regroupement qui va permettre de découvrir le schéma d'une source de données liées. En effet, il existe plusieurs algorithmes de clustering qui diffèrent selon plusieurs facteurs, tels que la manière dont les clusters sont produits, le type d'attributs qu'ils traitent, la capacité de traiter des données de haute dimension (grand nombre d'attributs), la dépendance de l'ordre d'arrivé des données, etc. Dans la section qui suit, nous présentons une analyse des algorithmes de clustering afin d'identifier celui qui répond au mieux aux caractéristiques des données liées sur le Web.

3.3 Analyse des algorithmes de clustering pour les données du Web sémantique

Dans cette section, nous nous intéressons à la façon dont le clustering peut être utilisé pour extraire un schéma à partir de données liées non structurées. Nous analysons tout particulièrement l'adéquation des algorithmes de clustering

avec les données du Web sémantique. Le clustering, aussi appelée segmentation ou classification automatique non supervisée [64], sert à diviser les objets (entités, individus, points, instances) d'une population (source de données, jeu de données) en groupes (clusters, classes, concepts), de telle sorte que les objets d'un même cluster soient similaires et les objets de clusters distincts soient dissimilaires. Au lieu d'examiner à chaque fois les objets qui se ressemblent, il suffit de le faire sur le représentant de leur groupe.

Le clustering procède par un apprentissage non supervisé, qui a pour but d'obtenir des informations sans aucune connaissance préalable. Ses applications sont nombreuses [65], en statistique, traitement d'image, intelligence artificielle, reconnaissance des formes ou encore en compression de données. Pour toutes ces raisons, il constitue une tâche importante dans le processus de fouille de données.

Le clustering peut être également utilisé pour extraire le schéma d'une source de données [31, 26, 43, 4, 66] en regroupant les entités : deux entités sont regroupées sur la base de leurs propriétés et non pas sur la base des valeurs de leurs propriétés. Comme le but du regroupement ici est de trouver la structure globale de la source de données, alors les entités partageant des propriétés communes sont regroupées en cluster, pour former les types du schéma. Dans cette section, notre objectif est d'identifier les algorithmes de clustering dont les propriétés répondent le mieux aux caractéristiques des données du Web sémantique.

Cette section est organisée comme suit. Nous analysons d'abord les caractéristiques des données du Web sémantique dans la section 3.3.1, ensuite nous discutons les familles principales d'algorithmes de clustering dans la section 3.3.2. Nous synthétisons notre étude dans la section 3.3.3.

3.3.1 Caractéristiques des données du Web sémantique influant sur le choix de l'algorithme

Une entité du Web sémantique est une ressource liée à plusieurs autres ressources. Chaque lien est une propriété dont la valeur est une ressource ou un littéral. Puisque le but du clustering ici est de regrouper les entités structurellement proches, alors il doit comparer des ensembles de propriétés et non pas leurs valeurs. Par conséquent, les données à traiter par le clustering sont de type catégoriel, non spatial et ne suivent aucune distribution.

Les instances de même type peuvent être décrites par des ensembles de propriétés très différents. L'algorithme de clustering doit pouvoir former des clusters de forme arbitraire afin d'autoriser le regroupement d'instances moins proches structurellement. Un autre aspect non négligeable des données liées est le fait qu'une ressource peut être de plusieurs types ; et par conséquent, appartenir à plusieurs clusters. Les données publiées sur le Web sont généralement en constante évolution, de grande taille et peuvent contenir des valeurs manquantes et aberrantes, car elles peuvent être peuplées de manières différentes : automatiquement par

l'extraction d'informations à partir de documents, comme *DBpedia*, ou manuellement par des acteurs différents. Le nombre de types que va avoir le schéma n'est pas connu à l'avance. Les données sont organisées en hiérarchie de types. Nous résumons les caractéristiques des données liées pour un traitement par clustering comme suit :

- Les données sont de type catégoriel, non spatial et ne suivent aucune distribution. L'algorithme de clustering choisi doit être adapté à ce type de données ;
- Deux données de même type peuvent être décrites par des ensembles de propriétés très différents. L'algorithme de clustering doit pouvoir construire des clusters de forme arbitraire ;
- Une données peut avoir plusieurs types, ce qui nécessite un clustering flou ;
- La source de données est en constante évolution et de grande taille, ce qui nécessite un processus incrémental et déterministe ;
- Le nombre de types n'est pas connu à l'avance, ce qui exclu les approches nécessitant le nombre de clusters en paramètre ;
- Les données peuvent contenir des valeurs manquantes ou aberrantes, ce qui nécessite un algorithme robuste au bruit ;
- Il peut y avoir un lien de hiérarchie entre les données. Il existe des algorithmes de clustering qui génèrent par construction des liens de hiérarchie entre les données.

3.3.2 Principales familles d'algorithmes de clustering

Les algorithmes de clustering varient selon plusieurs facteurs : la manière dont les clusters sont produits, le type d'attributs qu'ils traitent, la capacité de traiter une source de données volumineuse, la capacité de traiter des données de haute dimension ayant un grand nombre d'attributs, l'influence de l'ordre d'arrivé des données sur leur fonctionnement, etc. Selon ces critères et leurs combinaisons, il y a plusieurs familles d'algorithmes de clustering [64, 67, 68], dont les principales sont :

- **Algorithmes de partitionnement :** les algorithmes de partitionnement construisent différentes partitions, puis les évaluent selon certains critères. Tout d'abord, ils créent une partition initiale. Puis, à chaque itération, ils optimisent le partitionnement en déplaçant les objets d'un cluster à l'autre. D'après la représentation d'un cluster, les algorithmes de partitionnement sont répartis en deux groupes [67] : les algorithmes à base de K-Means [69] et les algorithmes à base de K-Medoïds [70]. Les algorithmes à base de K-Means représentent un cluster par un centroïde qui est un vecteur où chacune des composantes correspond à la valeur moyenne des objets du cluster. Les algorithmes à base de K-Medoïds représentent un cluster par un médoïde qui est un objet d'un cluster dont la distance avec les autres objets

de ce même cluster est minimale. L'avantage de cette famille d'algorithmes est leur complexité linéaire $o(n)$, car ils ne calculent pas la distance entre tous les objets, mais uniquement par rapport aux centres. Cependant, ils nécessitent de fixer le nombre de clusters voulus, ils ne sont pas déterministes et ils présentent une sensibilité au bruit. De plus, ils construisent des clusters de forme non arbitraire.
Les algorithmes de partitionnement ont pour objectif de minimiser la distance entre les objets d'un cluster et son centre, ce qui les rend rapides. L'algorithme k-means a été utilisé dans [31] pour l'extraction de schéma à partir de données semi-structurées. Cependant, l'utilisation de ce type d'algorithme de partitionnement requiert la connaissance *a priori* du nombre de clusters voulus. Ce qui n'est pas évident à définir, car on ne connaît pas *a priori* le nombre de types contenus dans la source de données.

- **Algorithmes basés sur la densité :** ces algorithmes considèrent les clusters comme étant des régions denses dans l'espace des objets. Un objet de l'espace est dense si le nombre de ses voisins dépasse un certain seuil fixé *a priori*. Ces algorithmes tentent alors d'identifier les clusters selon la densité des objets dans une région. Les objets sont alors groupés non pas sur la base d'une distance mais sur la base de la densité de voisinage si elle excède une certaine limite. Parmi les algorithmes les plus connus dans cette catégorie, nous pouvons citer [71, 68] : DBSCAN [11], OPTICS [72] et DENCLUE [73].
  Ces algorithmes sont déterministes, robustes au bruit et ils peuvent trouver des clusters de forme arbitraire. Cependant, les paramètres du rayon de voisinage et du nombre minimum d'objets dans le voisinage sont difficiles à définir, mais cela est moins complexe que le nombre de clusters. Les algorithmes proposés ne traitent pas les données catégorielles, mais leur idée de base peut être adaptée pour permettre leur traitement. En effet, on peut considérer la distance entre deux entités par rapport au nombre de propriétés différentes et calculer le voisinage sur cette base. Dans ces algorithmes, une donnée est affectée à un seul cluster et les liens de hiérarchie entre les données ne sont pas fournis.
- **Algorithmes hiérarchiques :** une hiérarchie peut être vue comme un ensemble de partitions emboîtées. Elle est généralement représentée par une structure arborescente. Les algorithmes hiérarchiques construisent les clusters graduellement sous la forme graphique d'un arbre hiérarchique. Il existe deux méthodes de segmentation hiérarchique : ascendante et descendante. Les algorithmes hiérarchiques ascendants considèrent, au départ, chaque objet comme un cluster. A chaque itération deux clusters sont fusionnés, si leur degré de similarité dépasse un certain seuil, afin de former un nouveau cluster. Les algorithmes hiérarchiques descendants procèdent par une méthode divisive. Ces algorithmes consistent, à partir d'un seul cluster regroupant

tous les objets, à diviser celui-ci en clusters plus raffinés. La segmentation hiérarchique ascendante est la plus utilisée. Ces méthodes sont itératives, le critère d'arrêt peut être le nombre de clusters désiré, le nombre d'itérations, le nombre minimum ou maximum d'objets dans chaque cluster, l'inertie, etc.
Les algorithmes hiérarchiques créent une décomposition hiérarchique de la source de données. Ce qui génère par construction des liens de hiérarchie entre les données, mais il ne s'agit pas de liens parfaitement sûrs mais seulement des possibilités. Les algorithmes hiérarchiques sont robustes au bruit et ils traitent différents types de données. Une approche qui s'appuie sur un clustering hiérarchique ascendant standard pour déduire des résumés structurels pour des données liées est proposée dans [26]. Une autre approche proposée dans [43] utilise également le clustering hiérarchique, mais pour construire une hiérarchie des types déclarés dans le jeu de données. Pour cela, les déclarations *rdf:type* des instances sont exploitées. Néanmoins, l'utilisation d'un algorithme hiérarchique est très coûteuse et non incrémentale. De plus, un objet ne peut pas être affecté à plusieurs types et le paramètre de coupure de l'arbre hiérarchique est difficile à définir.

- **Algorithmes statistiques :** ces algorithmes utilisent des méthodes statistiques telles que les probabilités pour le regroupement des données. Parmi les systèmes de clustering utilisant des modèles statistiques, nous citons l'algorithme COBWEB [33]. Celui-ci est un algorithme statistique hiérarchique incrémental. Il est décrit comme un algorithme conceptuel car il trouve des descriptions caractéristiques de chaque groupe. L'algorithme organise progressivement les objets en un arbre de classification. Chaque nœud dans un arbre de classification représente un cluster et il est étiqueté par des probabilités qui résument la distribution des valeurs des attributs des objets classés dans le nœud. Cependant, il est non déterministe car le résultat du clustering dépend de l'ordre d'arrivée des données.
  COBWEB est un algorithme statistique incrémental qui forme une hiérarchie pendant le processus de clustering, ce qui peut servir pour inférer des liens hiérarchiques. Le travail présenté dans [4], propose d'utiliser COBWEB pour la construction d'un dataguide approximatif pour un jeu de données semi-structurées. Néanmoins, cet algorithme n'est pas déterministe et un objet ne peut être affecté qu'à un seul cluster. De plus, le stockage de l'ensemble des instances dans la hiérarchie est coûteux.
- **Algorithmes utilisant les grilles :** le principe de cette famille d'algorithmes est de diviser l'espace de données en cellules. Les régions jugées denses, selon la valeur d'un seuil fixé *a priori*, sont fusionnées, et celles jugées peu denses permettent d'établir les frontières entre les cellules. Ce type d'algorithme est conçu pour des données spatiales (géographiques, localisées dans l'espace). Une cellule est un produit cartésien de sous intervalles d'at-

tributs de données. L'espace des objets est quantifié en un nombre fini de cellules qui forment une structure de grille sur laquelle toutes les opérations sont effectuées pour former les clusters. L'avantage principal de cette approche est son temps de traitement rapide, qui est généralement indépendant du nombre d'objets, mais qui dépend uniquement du nombre de cellules dans chaque dimension de l'espace quantifié. Avec une telle représentation des données, au lieu de faire le clustering sur l'espace de données, il est fait sur l'espace spatial en utilisant les informations statistiques des objets dans les cellules. Les algorithmes les plus connus dans cette catégorie sont : STING [74] et WaveCluster [75], qui sont adaptés pour les données spatiales uniquement.
Les algorithmes utilisant les grilles divisent l'espace de données en cellules. Les régions jugées denses selon la valeur d'un seuil fixé *a priori* sont fusionnées pour former les clusters. Ces algorithmes sont déterministes, robustes au bruit, incrémentaux et rapides. Cependant, ils sont proposés pour le traitement des données spatiales et ne permettent pas d'affecter une donnée à plusieurs types, ni de générer des liens de hiérarchie entre les données.

- **Algorithmes flous :** les algorithmes décrits ci-dessus produisent des clusters disjoints. Ce qui signifie qu'un objet donné ne peut appartenir qu'à un seul cluster. L'introduction de l'incertitude dans la tâche de clustering a conduit à la mise en place d'algorithmes qui utilisent les concepts de la logique floue dans leur procédure. Les algorithmes les plus connus dans cette catégorie sont FCM [76] et EM [77]. Ces algorithmes permettent d'affecter un objet à plusieurs clusters. Cependant, plusieurs itérations sont nécessaires pour avoir un bon résultat, et ils nécessitent la spécification *a priori* du nombre de clusters et du seuil de similarité. Pour l'algorithme EM, les données doivent en plus suivre un modèle de distribution, comme la loi normale.
  Les algorithmes flous peuvent affecter un objet à plusieurs clusters avec différents degrés d'appartenance. Cette caractéristique est très présente dans les données liées. En effet, une donnée peut être de plusieurs types, par exemple une entité représentant une personne peut être un auteur, un organisateur, etc. FCM est une extension probabiliste de k-means. Il est inadapté pour le traitement des données liées pour les mêmes raisons que celles discutées précédemment pour k-means, notamment le nombre de clusters nécessaire comme paramètre. EM fait l'hypothèse que les données suivent un modèle de distribution, ce qu'on ne peut pas supposer pour les données liées.
- **Algorithmes évolutionnistes :** les algorithmes évolutionnistes sont une famille d'algorithmes s'inspirant de la théorie de l'évolution pour résoudre des problèmes divers. Ils font ainsi évoluer un ensemble de solutions à un problème donné, dans l'optique de trouver les meilleurs résultats. Ce sont

des algorithmes stochastiques, car ils utilisent itérativement des processus aléatoires. Parmi ces algorithmes, nous citons l'algorithme Génétique [78]. Cet algorithme est appliqué après un algorithme de clustering (ex : k-means, hiérarchique) pour trouver les meilleurs représentants des clusters en tenant comptes des éventuelles mutations de la source de données.
L'algorithme génétique en soi ne forme pas les clusters, mais trouve le meilleur représentant pour chaque cluster formé, et cela en essayant de simuler les évolutions potentielles de la source de données. Ce qui est bien adapté aux données liées sur le Web, car leur évolution est fréquente.

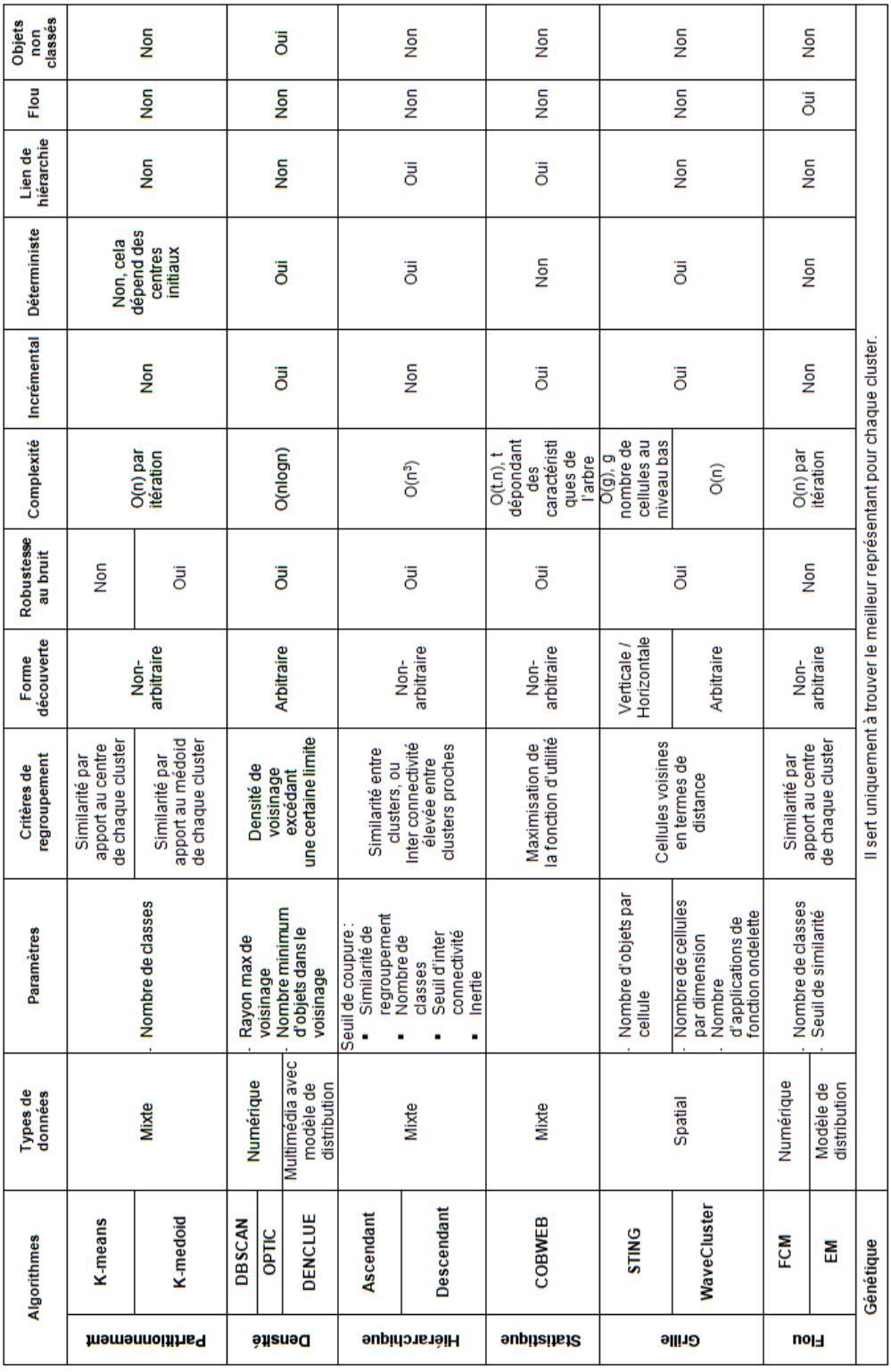
Le tableau 3.2 résume les caractéristiques de chaque algorithme décrit dans cette section. Nous comparons ces différents algorithmes en termes de : type de données traitées, paramètres requis, critère de regroupement, forme des clusters découverts, robustesse aux valeurs extrêmes, complexité, aspect incrémental, déterministe, génération de liens de hiérarchie, affectation d'un objet à plusieurs clusters, objets non classés.

3.3.3 Bilan sur l'adéquation des algorithmes de clustering aux données du Web sémantique

Les algorithmes de clustering présentés dans la section précédente ne sont pas réellement adaptés aux caractéristiques des données liées. Cependant, certains d'entre eux répondent à quelques exigences discutées dans la section 3.3.1.

Les algorithmes de partitionnement et les algorithmes flous ne sont pas adaptés car ils nécessitent le nombre de clusters comme paramètre, alors qu'on ne connaît pas le nombre de types du jeu de données *a priori*. Cependant, les algorithmes flous permettent d'attribuer plusieurs types à une entité, avec un degré d'appartenance, ce qui est l'une des principales caractéristiques des données liées. L'algorithme statistique COBWEB est incrémental et forme une hiérarchie. Cependant, il n'est pas déterministe. Les algorithmes hiérarchiques sont déterministes et fournissent des liens de hiérarchie entre les données. Cependant, ils ne peuvent pas découvrir des clusters de forme arbitraire, ils ne sont pas incrémentaux et leur principal limitation est leur coût. Les algorithmes basés sur la densité sont moins coûteux, mais ils ne forment pas les liens hiérarchiques.

Les algorithmes basés sur la densité sont les plus appropriés pour les données liées, car ils sont robustes au bruit, déterministes et ils ne nécessitent pas la spécification du nombre de clusters. De plus, ils trouvent des clusters de forme arbitraire : l'apport principal de cette famille d'algorithmes est de ne pas limiter la recherche uniquement aux clusters constitués d'objets très similaires, mais au contraire, d'autoriser la formation de clusters d'objets moins proches structurellement. Ceci est utile pour les sources de données où les entités sont décrites avec des ensembles de propriétés hétérogènes. Les algorithmes basés sur la densité font



Algorithmes		Types de données	Paramètres	Critères de regroupement	Forme découverte	Robustesse au bruit	Complexité	Incrémental	Déterministe	Lien de hiérarchie	Flou	Objets non classés
Partitionnement	K-means	Mixte	- Nombre de classes	Similarité par apport au centre de chaque cluster	Non-arbitraire	Non	O(n) par itération	Non	Non, cela dépend des centres initiaux	Non	Non	Non
	K-medoid			Similarité par apport au médoid de chaque cluster		Oui						
Densité	DBSCAN	Numérique	- Rayon max de voisinage - Nombre minimum d'objets dans le voisinage	Densité de voisinage excédant une certaine limite	Arbitraire	Oui	O(nlogn)	Oui	Oui	Non	Non	Oui
	OPTIC											
	DENCLUE	Multimédia avec modèle de distribution										
Hiérarchique	Ascendant	Mixte	Seuil de coupure : ▪ Similarité de regroupement ▪ Nombre de classes ▪ Seuil d'inter connectivité ▪ Inertie	Similarité entre clusters, ou Inter connectivité élevée entre clusters proches	Non-arbitraire	Oui	$O(n^3)$	Non	Oui	Oui	Non	Non
	Descendant											
Statistique	COBWEB	Mixte		Maximisation de la fonction d'utilité	Non-arbitraire	Oui	O(t.n), t dépondant des caractéristiques de l'arbre	Oui	Non	Oui	Non	Non
Grille	STING	Spatial	- Nombre d'objets par cellule	Cellules voisines en termes de distance	Verticale / Horizontale	Oui	O(g), g nombre de cellules au niveau bas	Oui	Oui	Non	Non	Non
	WaveCluster		- Nombre de cellules par dimension - Nombre d'applications de fonction ondelette		Arbitraire		O(n)					
Flou	FCM	Numérique	- Nombre de classes - Seuil de similarité	Similarité par apport au centre de chaque cluster	Non-arbitraire	Non	O(n) par itération	Non	Non	Non	Oui	Non
	EM	Modèle de distribution										
Génétique		Il sert uniquement à trouver le meilleur représentant pour chaque cluster.										

TABLE 3.2 – Tableau comparatif entre les principales familles d'algorithmes de clustering.

de cette notion le point central de leur processus, nous présentons leurs principes dans la section suivante, ainsi que leur adaptation pour la découverte du schéma d'une source de données liées.

3.4 Approche générale pour la découverte de schéma

Nous proposons une approche qui permet la découverte automatique du schéma implicite d'une source de données RDF. Cela consiste en la découverte des types et des liens entre eux. Nous proposons, dans la section 3.4.1, d'adapter l'algorithme de clustering DBscan pour regrouper les instances et ainsi découvrir les types implicites de la source de données. Nous présentons, ensuite, dans la section 3.4.2, le framework général de notre approche de découverte de schéma.

3.4.1 Utilisation de DBscan pour le regroupement

Pour découvrir les types contenus dans un jeu de données liées, nous proposons une approche fondée sur les principes d'un algorithme de clustering basé sur la densité. DBscan est l'algorithme initial de cette famille d'algorithmes. Il est approprié pour les données liées car il trouve des clusters de forme arbitraire, il est robuste au bruit et déterministe. De plus, le nombre de clusters n'est pas requis comme paramètre.

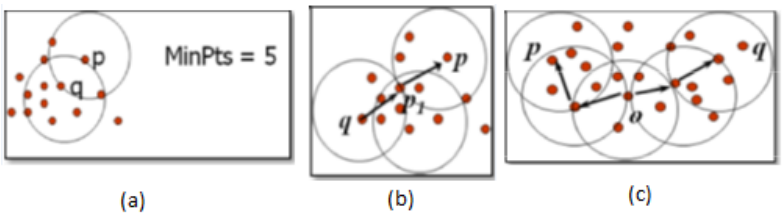


FIGURE 3.4 – Illustration de : *atteignabilité directe* (a), *atteignabilité* (b) et *cluster* (c) [11].

DBscan est un algorithme de clustering basé sur la densité. Dans notre contexte, nous sommes particulièrement intéressés par le concept d'atteignabilité dans cet algorithme, car il permet de construire des clusters de forme arbitraire. DBscan a deux paramètres : ε la valeur de similarité minimale pour deux entités à considérer comme voisines, $V_\varepsilon(q)$ l'ensemble des voisins d'une entité q et $MinPts$ le nombre minimum d'entités dans un voisinage. DBscan est fondé sur les concepts suivants [11] :

Définition (Atteignabilité [6] directe). Un objet p est directement densité-atteignable à partir d'un objet q (voir figure 3.4 (a)) si :

- $p \in V_\varepsilon(q)$;
- $|V_\varepsilon(q)| \geq MinPts$ (q est un noyau).

Définition (Atteignabilité). Un objet p est atteignable à partir d'un objet q (voir figure 3.4 (b)) s'il y a n objets $p_1, ..., p_n$, avec $p_1 = q$, $p_n = p$, tels que p_{i+1} est directement densité-atteignable à partir de l'objet p_i pour $1 < i < n$.

Définition (Cluster). Un cluster est l'ensemble maximal d'objets connectés en considérant qu'un objet p est connecté à q s'il existe un objet o tel que p et q sont tous les deux atteignables à partir de o (voir figure 3.4 (c)).

Le concept d'atteignabilité dans DBscan permet de : (i) regrouper des données sans avoir à définir le nombre de clusters car il s'agit d'un clustering par propagation ; (ii) découvrir des clusters de forme arbitraire, ce qui est important pour les données liées où des entités similaires peuvent être décrites par des ensembles de propriétés différents ; (iii) regrouper les données en une seule itération de manière déterministe et (iv) détecter le bruit en considérant les objets non atteignables comme du bruit.

Pour maximiser l'atteignabilité, nous devons maximiser le concept d'atteignabilité directe en minimisant le paramètre $MinPts$. Ce paramètre représente le nombre minimum requis d'entités dans le voisinage pour qu'une entité soit un noyau et, dans notre contexte, pour générer un type ; il permet d'exclure les données bruitées. Comme deux entités du même type peuvent être décrites par des ensembles de propriétés différents, nous considérons dans notre approche qu'il suffit qu'une entité soit semblable à une autre entité pour être considérée comme non bruitée et pour former un type. Pour cette raison et pour maximiser l'atteignabilité, nous fixons $MinPts$ à 1. Cela implique que chaque objet dans le jeu de données est un noyau s'il a au moins un voisin, et il suffit que deux objets p et q soient semblables pour qu'ils soient mutuellement atteignables.

DBscan nécessite de fixer le paramètre de seuil de similarité, qui est très important pour la qualité des types résultants. Ce paramètre n'est pas évident à définir. Une autre exigence que DBscan ne satisfait pas dans notre contexte est qu'il ne peut pas assigner plusieurs types à une entité car les clusters découverts sont disjoints. Il faudrait également compléter les types découverts par DBscan en inférant des liens entre eux. Dans la section suivante, nous présentons notre approche générale qui adapte DBscan pour la découverte du schéma d'un jeu de données liées.

6. En anglais *Reachability* (source Wikipedia)

3.4.2 Framework général

Notre approche générale pour la découverte automatique du schéma d'une source de données RDF est représentée dans la figure 3.5. Nous proposons de découvrir le schéma d'une source de données en deux étapes principales :

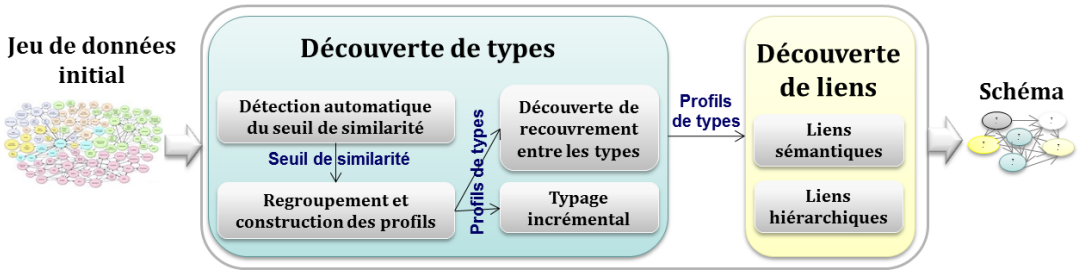


FIGURE 3.5 – Approche générale de découverte de schéma.

- **Découverte des types :** étant donné une source de données RDF sans aucune information relative au schéma, nous proposons de regrouper les données selon la similarité de leurs propriétés respectives. Nous adaptons pour cela un algorithme de clustering enrichi par les fonctionnalités suivantes :
  - *Détection automatique du seuil de similarité :* pour regrouper des entités selon leur similarité, il est nécessaire de fixer un seuil de similarité. Comme nous ne connaissons pas le nombre de types contenus dans le jeu de données, il n'est pas possible de connaître *a priori* le nombre de groupes voulu. Nous proposons donc une méthode automatique qui détecte un seuil de similarité pour le regroupement selon l'hétérogénéité des données.
  - *Regroupement et construction des profils de types :* nous regroupons les entités selon leur similarité structurelle en respectant le seuil de similarité détecté. Durant le processus de regroupement, nous généralisons les propriétés de chaque groupe pour avoir une description structurelle globale de la source de données.
  - *Découverte de recouvrement entre les types :* une entité RDF peut avoir plusieurs types, par exemple, l'entité $P1$ de la figure 3.1, est de type *Author* et *Organizer*. Pour cela, nous proposons une méthode qui utilise les profils des types pour découvrir les recouvrements entre les types.
  - *Typage de nouvelles entités :* une source de données du Web est généralement en constante évolution, comme *DBpedia* qui est régulièrement mise à jour à partir des pages de *Wikipedia*. Pour cela, nous proposons une approche pour assurer l'évolution du schéma suite à l'évolution du contenu de la source de données, en permettant le typage de nouvelles entités et la création de nouveaux types.

- **Génération de liens :** nous proposons de découvrir deux types de liens :
  - *Liens sémantiques :* ils correspondent à des propriétés définies par l'utilisateur et qui peuvent être généralisées par des liens entre deux groupes d'entités de même type. Ces liens peuvent être clairement définis dans le jeu de données à travers les déclarations $rdfs{:}range$ et $rdfs{:}domain$. Cependant, ces déclarations sont souvent incomplètes ou manquantes. Ces liens sont très utiles pour exploiter le jeu de données. En effet, le schéma décrit dans la figure 3.3, montre par exemple que le type *Presentation* est lié au type *Conference* par le lien *isRoleAt*. Ce qui peut être utile pour formuler les requêtes. Les liens entre les types ne sont pas évidents à trouver : nous pouvons parcourir le jeu de données et insérer un lien entre deux types à chaque fois que deux entités de ces deux types sont liées par ce lien. Cependant, appliquer cette méthode à un jeu de données volumineux serait très coûteux.
  - *Liens hiérarchiques :* cela représente des liens $rdfs{:}subClassOf$, comme dans la figure 3.3 entre les types *Organizer* et *Person* et entre les types *Author* et *Person*. Ces liens hiérarchiques ne sont pas exprimés entre les entités mais entre les types. Pour établir les liens entre les types découverts, nous proposons d'analyser le profil de chaque type.

Dans ce qui suit, nous détaillons chacun des aspects présentés ci-dessus, et nous montrons comment générer un schéma pour une source de données et fournir ainsi une vue globale du jeu de données à partir de ses entités sans qu'aucune déclaration sur le schéma ne soit nécessaire, et cela d'une façon automatique et évolutive.

3.5 Découverte des types

Pour découvrir les types contenus dans un jeu de données liées, nous proposons de regrouper les entités selon leur similarité structurelle en adaptant l'algorithme de clustering DBscan. Selon l'analyse faite dans la section 3.3, les exigences pour notre approche de découverte du schéma d'une source de données sont les suivantes : (i) les clusters détectés doivent être de forme arbitraire, car les entités de même type peuvent être décrites par des propriétés hétérogènes (ii) le nombre de clusters en paramètre ne peut pas être fourni, car le nombre de types dans la source de données est inconnu (iii) l'approche de regroupement doit être robuste au bruit car la source de données peut être bruitée (vi) le regroupement doit autoriser le recouvrement entre les clusters, car une entité peut avoir plusieurs types et (v) l'approche doit permettre de trouver le(s) type(s) d'une nouvelle entité, car la source de données peut évoluer avec l'ajout de nouvelles entités.

DBscan est approprié pour le regroupement des données liées, comme

discuté dans la section 3.3.3, car il trouve des clusters de forme arbitraire, il est robuste au bruit et déterministe. De plus, le nombre de clusters n'est pas requis comme paramètre. Cependant, DBscan requiert la spécification du seuil de similarité pour le clustering. Ce paramètre est très important pour la qualité des types obtenus. Une autre de nos exigences que l'algorithme DBscan ne satisfait pas est qu'il ne peut pas assigner plusieurs types à une entité car les clusters découverts sont disjoints. Des algorithmes de clustering flou tels que FCM [76] ou EM [77] pourraient être utilisés pour attribuer plusieurs types à une entité. Cependant, ils exigent le nombre de clusters car leur critère de regroupement est la similarité entre une entité et le centre de chaque cluster. A mesure que la source de données évolue et que l'on ajoute de nouvelles entités, un problème qui apparaît est la détermination des types d'une entité entrante.

Nos contributions pour la découverte automatique des types sont les suivantes :

- L'adaptation d'un algorithme de clustering basé sur la densité pour la découverte des types dans un jeu de données liées, avec la construction d'un profil probabiliste qui caractérise chaque type ;
- La détection des recouvrements entre les types découverts ce qui permet à une entité d'avoir plusieurs types ;
- La détection automatique du paramètre du seuil de similarité pour le regroupement des données ;
- La découverte de(s) type(s) d'une nouvelle entité ajoutée au jeu de données.

Nous décrivons dans cette section, les algorithmes qui sous-tendent notre approche. Celle-ci est fondée sur l'algorithme de clustering DBscan qui sera présenté dans la section 3.5.1. La découverte des recouvrements entre les types est décrite dans la section 3.5.2. L'approche est aussi auto-adaptative grâce à un algorithme de détection automatique du seuil de similarité présenté dans la section 3.5.3. L'aspect incrémental de l'approche est abordé dans la section 3.5.4

3.5.1 Regroupement des entités

L'algorithme de clustering DBscan a deux paramètres : ε, représentant la valeur minimum de similarité pour que deux entités soient voisines, que nous proposons de déduire de manière automatique, et $MinPts$, qui représente le nombre minimal requis d'entités dans le voisinage d'une entité pour que cette dernière soit un noyau [11] et par conséquent soit à l'origine de la création d'un type ; $MinPts$ est utilisé pour exclure les entités bruitées. Pour maximiser l'atteignabilité, nous fixons $MinPts$ à 1, comme expliqué dans la section 3.4.1. Pour mesurer la similarité entre deux ensembles de propriétés $e.P_U$ et $e'.P_U$ décrivant deux entités e et e' respectivement, nous utilisons la mesure de similarité de Jaccard, définie ci-après.

Algorithm 1: DBscan avec génération des profils de types

Input : D_U, ε, $MinPts$
Output: $ensProfilsType$, ensemble des données D_U avec des types

```
ensProfilsType ← ∅;
classe ← 0;
while ∃ non marquée e.P_U ∈ D_U do
    marquer e.P_U;
    ensVoisins ← TrouverVoisins(D_U, e.P_U, ε);
    if |ensVoisins| ≥ MinPts then
        classe++;
        ensProfilsType ← ensProfilsType ∪ EtendreCluster(D_U,e.P_U,
          MinPts, ε, classe, ensVoisins);
    end
end
```

Définition (Similarité de Jaccard). La similarité de Jaccard représente le rapport entre la taille de l'intersection des ensembles de propriétés considérés et la cardinalité de leur union. Cette mesure de similarité est adaptée pour évaluer la similarité entre des ensembles de taille différente. La similarité de Jaccard entre deux ensembles de propriétés $e.P_U$ et $e'.P_U$ décrivant deux entités e et e' respectivement est décrite dans la formule 3.1 :

$$Similarité(e, e') = \frac{|e.P_U \cap e'.P_U|}{|e.P_U \cup e'.P_U|} \tag{3.1}$$

En plus de la découverte de l'ensemble des types, nous décrivons chaque type par un profil. Nous allons utiliser les profils pour : (i) découvrir les recouvrements entre les types, ce qui permet à une entité d'avoir plusieurs types ; (ii) permettre de typer une nouvelle entité ; (iii) découvrir les liens sémantiques et (iv) découvrir les liens hiérarchiques entre les types. Un profil consiste en un ensemble de propriétés où chaque propriété est associée à une probabilité. Nous définissons le profil d'un type comme suit.

Définition (Profil d'un type). Un profil de type est un ensemble de propriétés où chaque propriété est associée à une probabilité. Le profil correspondant à un type T_i est noté $TP_i = \{(p_{i1}, \alpha_{i1}), ..., (p_{in}, \alpha_{in})\}$, où chaque p_{ij} représente une propriété et où chaque α_{ij} représente la probabilité pour une entité de T_i d'avoir la propriété P_{ij}. Cette probabilité est évaluée comme le nombre d'entités de T_i ayant la propriété p_{ij} sur le nombre total d'entités dans T_i.

Dans la figure 3.1, le profil du type *Université* est : $\{(\overrightarrow{name}, 1), (\overrightarrow{website}, 0.5),$

$(\overleftarrow{affiliation}, 1)\}$. La probabilité de la propriété *website* est de 0.5 car cette propriété est définie pour la moitié des entités de ce type. En effet, cette propriété est définie pour l'entité *MIT* mais pas pour l'entité *UVSQ*.

Algorithm 2: Expand Cluster

Input : D_U, $e.P_U$, $MinPts$, ε, $class$, $ensVoisins$
Output: TP_{classe}

```
1  nbEntitiesInClass ← 0;
2  Profil de type de la classe TP_classe ← ∅;
3  Ajouter classe aux types de e ;
4  MettreAJourProfil(TP_classe, classe, e.P_U, nbEntitiesInClass);
5  nbEntitiesInClass++;
6  while ∃ e'.P_U ∈ ensVoisins do
7      if non marquée e'.P_U then
8          marquer e'.P_U;
9          ensVoisins' ← TrouverVoisins(D_U, e'.P_U, ε);
10         if |ensVoisins'| ≥ MinPts then
11             ensVoisins ← ensVoisins ∪ ensVoisins';
12         end
13     end
14     Ajouter classe aux types de e' ;
15     MettreAJourProfil(TP_classe, classe, e'.P_U, nbEntitiesInClass);
16     nbEntitiesInClass++;
17 end
```

L'algorithme DBscan construit un cluster en cherchant d'abord un objet noyau puis rassemble tous les objets atteignables à partir de ce noyau. Afin de regrouper des entités similaires et construire le profil de chaque type, nous avons adapté l'algorithme DBscan (**algorithme 1**) qui requiert en entrée l'ensemble des propriétés utilisateur des données D_U, le rayon de voisinage ε et le nombre minimum de voisins d'un noyau $MinPts$. L'ensemble des profils des types est d'abord initialisé à vide, et l'identifiant des classes à 0. Notre algorithme parcourt l'ensemble D_U et pour chaque objet non encore traité (marqué), il construit son ensemble d'objets voisins en utilisant la fonction **TrouverVoisins**. Chaque nouvelle entité e ayant un nombre de voisins supérieur à $MinPts$ est un noyau qui sera à l'origine de la création d'un type. Ce type sera étendu en utilisant la fonction **EtendreCluster**, définie dans l'**algorithme 2**. L'algorithme 1 retourne l'ensemble D_U avec le type d'appartenance de chaque entité, ainsi que le profil de chaque type.

La fonction **EtendreCluster**, définie dans l'**algorithme 2** permet de trouver toutes les entités appartenant au même type que l'entité courante e. Elle prend en entrée l'ensemble des propriétés utilisateur des données D_U, l'ensemble des propriétés utilisateur qui décrivent l'entité e, le nombre minimum de voisin d'un

Algorithm 3: Mettre à jour profil cluster

Input : TP_{classe}, $classe$, $e.P_U$, $nbEntitiesInClass$
Output: TP_{classe}

```
1 for each propriété p ∈ e.P_U do
2     if p est dans TP_classe avec α sa probabilité then
3         α ← ((α × nbEntitiesInClass) + 1) / (nbEntitiesInClass + 1);
4         Mettre à jour α dans TP_classe;
5     else
6         TP_classe ← TP_classe ∪ (p, 1/(nbEntitiesInClass + 1));
7     end
8 end
```

noyau $MinPts$, le rayon de voisinage ε, l'identifiant de la classe de e et l'ensemble des voisins de e. Le nombre d'entités dans la classe est d'abord initialisé à 0, et le profil du type à vide. L'identifiant de la classe courante est ajouté aux types de l'entité e, et le profil de type est mis à jour en utilisant la fonction **MettreA-JourProfil** décrite dans l'**algorithme 3**. Cette fonction ajoute les propriétés de e si elles ne sont pas déjà dans le profil et recalcule les probabilités des propriétés. Le nombre d'entités de la classe est incrémenté. Chaque entité e' appartenant à l'ensemble des voisins de e, est marquée si elle ne l'est pas déjà, et son ensemble de voisins est trouvé avec la fonction **TrouverVoisins**. Cette fonction permet de retourner pour une entité donnée toutes les entités ayant une distance inférieure à ε. Chaque entité e' qui a un nombre de voisins supérieur à $MinPts$ est un noyau, et son ensemble de voisins est ajouté à l'ensemble de voisins de e, afin d'étendre le cluster courant à toutes les entités atteignables par e. L'entité e' est ajoutée à la classe courante même si elle n'est pas un noyau, et le profil de type est mis à jour avec l'entité e' en utilisant la fonction **MettreAJourProfil** décrite dans l'**algorithme 3**. Le nombre d'entités de la classe est incrémenté. La fonction **EtendreCluster**, définie dans l'**algorithme 2** retourne le profil du type construit.

La fonction **MettreAJourProfil** décrite dans l'**algorithme 3** prend en entrée : le profil courant du type TP_{class}, l'ensemble des propriétés utilisateur qui décrivent l'entité e nouvellement ajoutée dans la classe, ainsi que le nombre d'entités de la classe. Pour chaque propriété utilisateur p décrivant l'entité e, si p est présente dans le profil de type TP_{class}, alors sa probabilité α est mise à jour pour considérer cette nouvelle entité e. Si p n'est pas présente dans le profil de type TP_{class}, alors elle est ajoutée avec sa probabilité.

La figure 3.6 représente le résultat du regroupement appliqué sur le jeu de données de la figure 3.1. Les entités sont bien regroupées selon leur similarité structurelle. Cependant, chaque entité appartient à un cluster unique, ce qui implique qu'elle n'est affectée qu'à un seul type. Ceci est dû au fait que l'algorithme

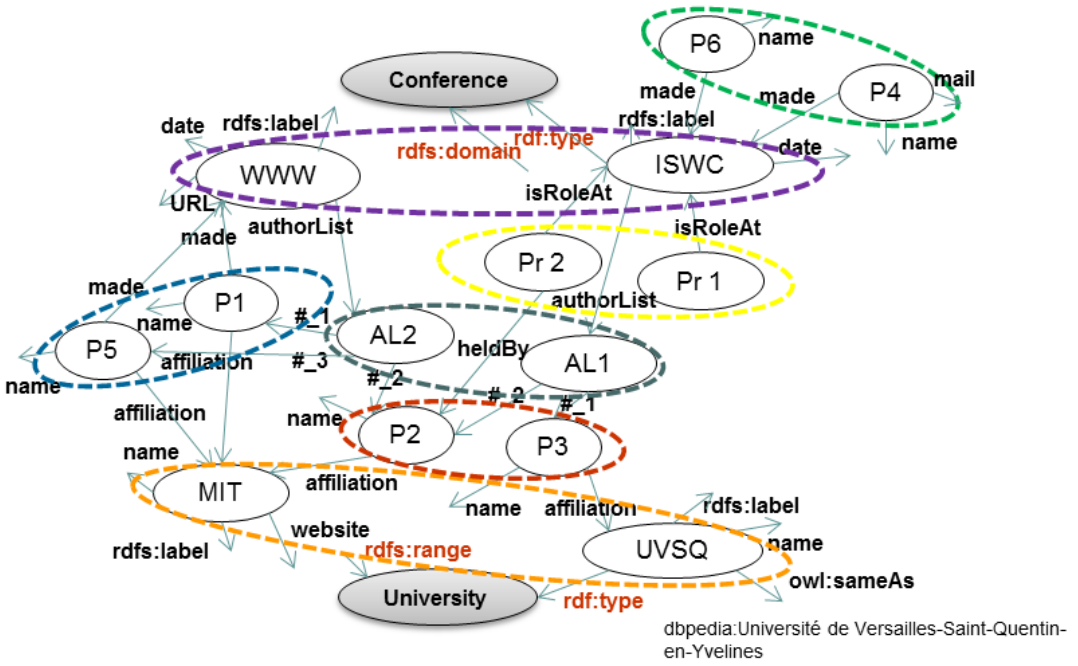


FIGURE 3.6 – Résultat de clusters disjoints du jeu de données de la figure 3.1.

de clustering DBscan ne permet pas d'avoir des clusters qui se recouvrent.

La complexité de DBscan est de $O(n^2)$, pour un jeu de données avec n entités. Dans [11], une extension de l'algorithme est proposée pour réduire la complexité de DBscan à $O(n * log(n))$ en utilisant des méthodes d'accès spatial telles que R*-trees [79].

Un certain nombre d'exigences pour le regroupement des données du Web sémantique ne sont pas satisfaites par DBscan : (i) une entité peut avoir plusieurs types, (ii) la détection automatique du seuil de similarité selon les entités du jeu de données et (vi) la prise en considération de l'évolution de la source de données avec l'ajout de nouvelles entités. Nous présentons dans les sections suivantes des propositions pour répondre à ces exigences.

3.5.2 Typage multiple d'une entité

Une caractéristique importante d'une source de données RDF(S)/OWL est qu'une entité peut avoir plusieurs types [40]. Un algorithme de clustering flou comme FCM ou EM pourrait être utilisé pour affecter plusieurs types à une entité. Cependant, il nécessite le nombre de clusters comme paramètre, ce qui ne peut être défini *a priori* dans notre contexte.

DBscan fournit un ensemble de clusters disjoints, mais une entité peut avoir plusieurs types. Notre problème est de savoir comment attribuer plusieurs types à une entité ?

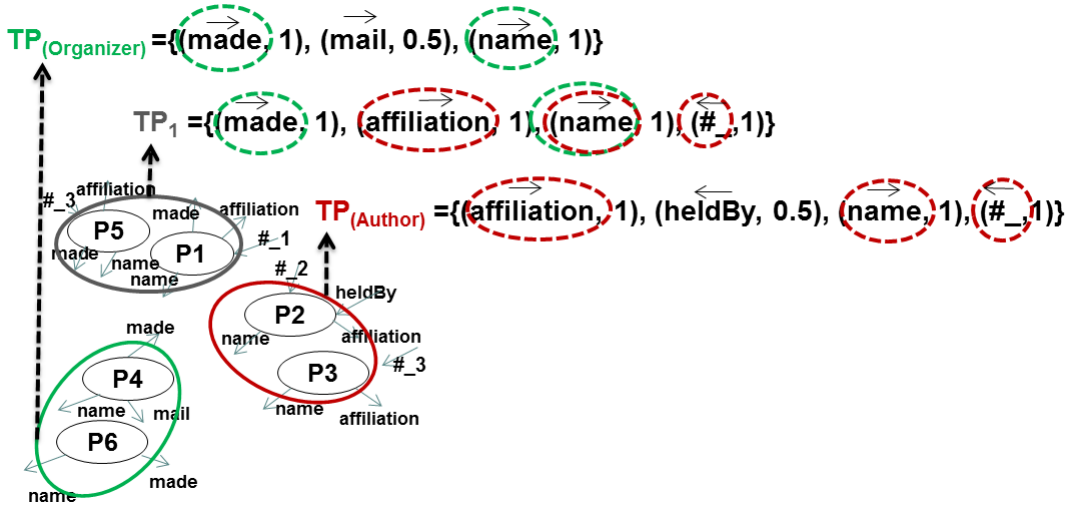


FIGURE 3.7 – Clusters disjoints avec leurs profils

Notre intuition est qu'une entité appartient à un type donné si elle est décrite par les propriétés de ce type. Cependant, les entités qui appartiennent à un type ne sont pas forcément décrites par toutes les propriétés de ce type. Dans le profil de type, chaque propriété est associée à une probabilité reflétant le nombre d'entités du type ayant cette propriété sur le nombre total d'entités du type. Nous proposons pour cela d'introduire la notion de propriété forte d'un type, qui est une propriété dans le profil du type pour laquelle la probabilité de décrire une entité du type est élevée. Nous considérons qu'une entité appartient à un type donné si elle est décrite par toutes les propriétés fortes de ce type. A l'issue de l'exécution de l'**algorithme 1**, des clusters disjoints avec leurs profils associés sont découverts. Nous proposons d'analyser le profil de chaque cluster pour générer des clusters qui se recouvrent.

Nous rappelons qu'un profil de type est un ensemble de propriétés où chaque propriété est associée à une probabilité. Le profil correspondant à un type T_i est noté $TP_i = ((p_{i1}, \alpha_{i1}), ..., (p_{in}, \alpha_{in}))$, où chaque p_{ij} représente une propriété et où chaque α_{ij} représente la probabilité pour une entité de T_i d'avoir la propriété P_{ij}. Le profil d'un type est construit progressivement lors du processus de clustering pour représenter la structure canonique de ce type. Nous définissons une propriété forte comme suit.

Définition (Propriété forte). Une propriété p est une propriété forte pour un type T_i étant donné un seuil θ, si $(p, \alpha) \in TP_i$ et $\alpha \geq \theta$.

La figure 3.7 montre un exemple de clusters disjoints et les profils de types correspondants. Considérons dans notre exemple que $\theta = 1$, on peut voir que le profil TP_1 possède toutes les propriétés fortes du profil du type *Author*, qui sont : *name*, *affiliation* et #_ qui représente l'appartenance à une liste d'auteurs. Ce

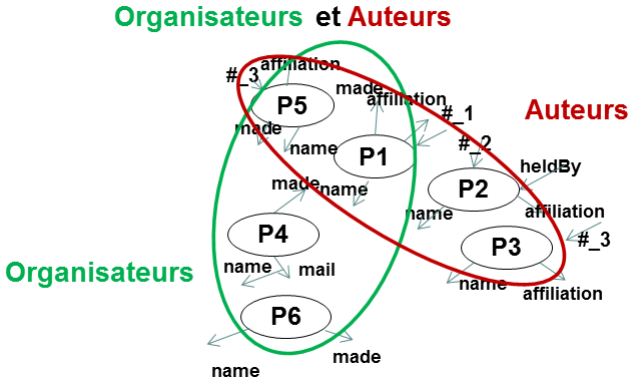


FIGURE 3.8 – Clusters en recouvrement

qui signifie que les entités de TP_1 sont également de type *Author*. Nous pouvons également observer que TP_1 a toutes les propriétés fortes du profil *Organizer*, qui sont : *name* et *made*. Ce qui signifie que les entités de TP_1 sont également de type *Organizer*. De la même manière, nous comparons toutes les paires de profils fournies par notre algorithme de clustering (**algorithme 1**). Comme décrit dans l'**algorithme 4**, à chaque fois que toutes les propriétés fortes d'un profil TP_i sont trouvées dans un autre profil TP_j, le type T_i est ajouté aux entités du type T_j. La figure 3.8 représente le résultat de cet algorithme de découverte de typage multiple appliqué aux données de la figure 3.7. Nous pouvons voir que le recouvrement entre les clusters contenant des auteurs et des organisateurs décrit le fait que les entités p_1 et p_5 possèdent à la fois les types *Author* et *Organizer*.

Algorithm 4: Affectation de plusieurs types à une entité

Input : $typeProfileSet$
Output: typage multiple des entités

```
for ∀ TP_i ∈ typeProfileSet do
    for ∀ TP_j ∈ typeProfileSet with i! = j do
        if ∀(p, α) dans TP_i, α ≥ θ : (p, α) dans TP_j then
            Ajouter T_i comme type pour les entités du cluster j;
        end
    end
end
```

L'**algorithme 4** présente le processus de découverte des recouvrements entre les clusters en utilisant les profils de type. Chaque paire de profils de type sont comparées pour détecter l'inclusion possible de propriétés fortes. Considérant le type T_i décrit par le profil TP_i, si chaque propriété forte p de TP_i appartient à un autre profil de type TP_j, alors le type T_i est un type pour les entités du cluster j.

L'algorithme 2 compare chaque paire de profils de type ; sa complexité est de $O(k^2)$, où k est le nombre de types dans un cluster et n le nombre d'entités dans le jeu de données, avec $k < n/2$, car le nombre minimum d'entités pour former un type est de 2.

3.5.3 Détection automatiquement du seuil de similarité

DBscan requiert la valeur du seuil de similarité ε, représentant la valeur de similarité minimale pour deux entités à considérer comme voisines. Ce paramètre n'est pas facile à définir : si on choisit un seuil de similarité faible, les entités de différents types peuvent être regroupées et des entités bruitées peuvent être affectées incorrectement à un cluster, comme illustré dans la figure 3.9 (a) ; en revanche, si on choisit un seuil de similarité élevé, des entités de même type ne seront pas nécessairement regroupées et des entités qui ne sont pas bruitées ne seront pas affectées à un cluster, comme illustré dans la figure 3.9 (b). La figure 3.9 (c) montre le résultat d'un clustering avec un seuil de similarité optimal. La valeur d'un seuil de similarité optimal permet de grouper des entités de même type et d'exclure les entités bruitées.

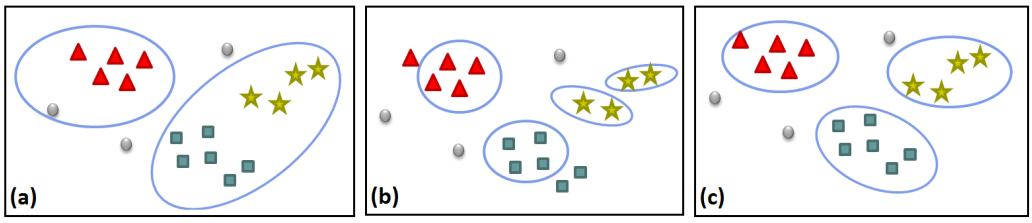


FIGURE 3.9 – Résultats du clustering avec un seuil de similarité faible (a), élevé (b) et optimal (c)

La figure 3.10 donne l'intuition derrière notre approche pour détecter automatiquement le seuil de similarité. Nous pouvons observer que pour les entités qui sont regroupées, la distance entre une entité et son plus proche voisin est petite, alors que la distance entre une entité bruitée et son plus proche voisin est importante. On peut considérer que la plus petite distance entre une entité bruitée et son plus proche voisin donne une idée du seuil de similarité. En d'autres termes, la plus grande similarité entre une entité non bruitée et son plus proche voisin est le seuil de similarité. Le problème qui se pose est comment identifier cette frontière ? Dans notre approche, nous essayons de déterminer l'entité dont la distance avec son plus proche voisin augmente de façon significative. Cela, afin de détecter le seuil de similarité pour un jeu de données.

Pour mesurer la similarité entre deux ensembles de propriétés $e.P_U$ et $e'.P_U$ décrivant deux entités e et e' respectivement, nous utilisons la mesure de similarité de Jaccard, définie précédemment dans la formule 3.1.

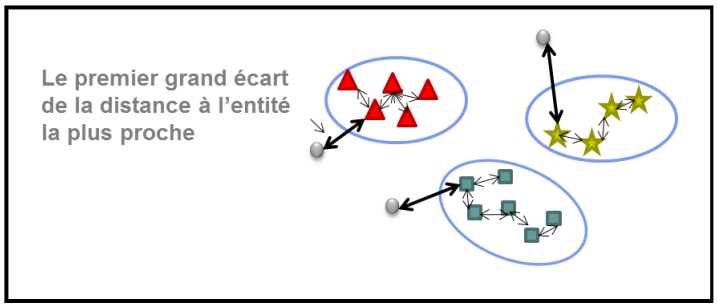


FIGURE 3.10 – Détection du seuil de similarité en se basant sur la distance au plus proche voisin.

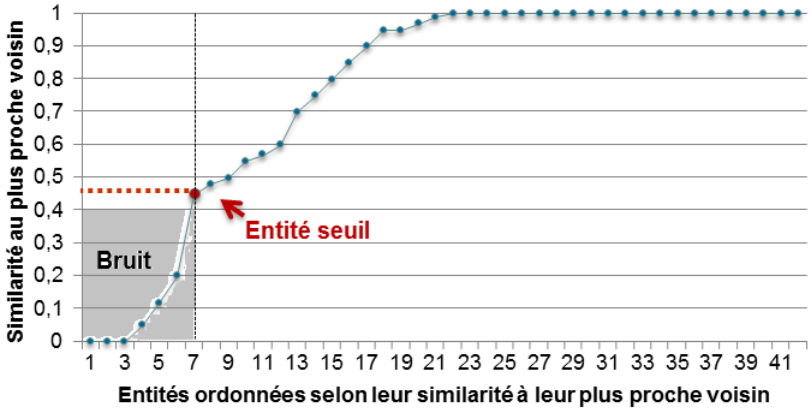


FIGURE 3.11 – Détection automatique du seuil de similarité.

Pour détecter quand la distance entre une entité e et son plus proche voisin augmente de façon considérable, nous proposons d'ordonner les entités en fonction de leur similarité croissante avec leur plus proche voisin $\delta(e)$ jusqu'à ce qu'il n'y ait plus d'entités à ordonner ou jusqu'à ce que la similarité avec le voisin le plus proche atteigne *1*, comme représenté dans la figure 3.11. Nous considérons pour chaque paire d'entités, la différence entre leurs similarités respectives par rapport à leur plus proche voisin. L'entité pour laquelle cette différence est maximale est appelée *entité seuil*, et la valeur de similarité de cette entité par rapport à son plus proche voisin sera le *seuil de similarité estimé* ε. Par exemple, dans la figure 3.11, l'*entité seuil* est l'entité 7, et ε est de 0.45. Toutes les entités qui précédent l'*entité seuil* représentent le bruit. Nous définissons l'*entité seuil* et le *seuil de similarité estimé* comme suit.

Définition (Entité seuil). Étant donné un ensemble d'entités ordonnées selon leur similarité croissante par rapport à leur plus proche voisin $E = \{e_1, ..., e_i, e_{i+1}, ...e_n\}$ et $\delta(e_i)$ qui représente la similarité d'une entité e_i avec son plus proche voisin ; l'entité seuil e_t est une entité de E ayant le plus grand écart

Algorithm 5: Détection automatique du seuil de similarité

Input : D_U
Output: ε

```
maxGap = 0;
for e_i.P_U ∈ D_U do
    Calculer sa similarité à son plus proche voisin comme δ(e_i.P_U);
end
while ( ∃ non marquée e_i.P_U ∈ D_U ∧ δ(e_i.P_U)! = 1 ) do
    Ordonner e_i.P_U dans ensEntitsOrdonns selon la similarité croissante à
     leur plus proche voisin ;
    Marquer e_i.P_U ;
end
while ∃ e_{j+1}.P_U ∈ ensEntitsOrdonns do
    cart = δ(e_{j+1}.P_U) - δ(e_j.P_U);
    if maxEcart < cart then
        maxEcart = cart;
        ε = δ(e_{j+1}.P_U);
    end
end
```

entre sa similarité avec son voisin le plus proche et celle de l'entité qui la précède comme suit :

$$\delta(e_t) - \delta(e_{t-1}) = Max_{i=1}^{n}(\delta(e_{i+1}) - \delta(e_i)) \tag{3.2}$$

Définition (Seuil de similarité estimé). Étant donnée une entité seuil e_t, le seuil de similarité estimé est la similarité de e_t par rapport à son plus proche voisin $\delta(e_t)$.

Notre approche pour la détection automatique du seuil de similarité est présentée dans l'**algorithme 5**. Pour chaque entité, nous calculons sa similarité avec son plus proche voisin. Ensuite, les entités sont ordonnées selon leur similarité croissante à leur plus proche voisin jusqu'à ce qu'il n'y ait plus d'entités à ordonner ou jusqu'à ce que la similarité avec le plus proche voisin atteigne *1*. Nous calculons l'écart entre la similarité au plus proche voisin de chaque entité successive. Nous considérons la similarité avec le plus proche voisin de l'entité ayant le plus grand écart avec la similarité de l'entité précédente comme le seuil de similarité ε.

La complexité du calcul de la similarité entre chaque paire d'entités pour trouver le voisin le plus proche et ainsi détecter le seuil de similarité optimal est $O(n^2)$, où n est le nombre d'entités. Cependant, le calcul de la similarité d'une

entité donnée avec toutes les autres pourrait être parallélisé. Cela réduirait la complexité pour chaque entité à $O(n)$, avec n traitements parallèles.

3.5.4 Typage incrémental des entités

Une source de données liées peut évoluer et de nouvelles entités peuvent y être ajoutées. Étant donné un ensemble d'entités regroupées en types, le problème qui nous intéresse, dans cette section, est de trouver le(s) type(s) d'une nouvelle entité.

Une nouvelle entité peut appartenir à un ou plusieurs clusters existants ; elle peut également former un nouveau cluster avec une entité non encore typée, ou elle peut être considérée comme du bruit. Une entité est considérée comme du bruit si elle n'a pas de voisins. Si une nouvelle entité est similaire à d'autres entités bruitées dans la source de données, elles formeront un nouveau cluster avec toutes les entités similaires. Dans notre contexte, les clusters reflètent les types de données. Une source de données concerne généralement un domaine spécifique comme la source de données *Conference* qui contient des données décrivant des articles, des auteurs, des organisateurs, etc. Nous considérons que les types découverts dans la source de données n'ont pas à être redéfinis chaque fois que la source de données évolue. Cependant, de nouveaux types peuvent être générés. Par conséquent, ce que nous entendons par typage incrémental est orienté vers le typage de nouvelles entités, et non vers la redéfinition des clusters existants. Notre objectif est d'assigner à une nouvelle entité un ou plusieurs types existants, ou éventuellement de générer un nouveau type à partir de cette entité.

Dans [80], une version incrémentale de DBscan a été proposée. L'approche identifie l'ensemble d'objets impactés par l'entité insérée ou supprimée et elle ré-exécute DBscan sur cet ensemble. Les résultats intermédiaires du clustering précédant sont conservés en mémoire, comme les connexions entre les objets. L'objectif principal de cette approche est de suivre l'évolution des clusters. Toutefois, elle est limitée par la taille de la mémoire où elle stocke tous les résultats intermédiaires du clustering précédent. De plus, comme DBscan, elle ne peut pas affecter plusieurs types à une entité. Elle nécessite également le seuil de similarité comme paramètre. La suppression d'une entité peut conduire à la disparition d'un cluster d'entités, et ces entités deviendront alors du bruit. Dans notre contexte, un cluster reflète un type. Après la découverte d'un type dans un jeu de données, il n'est pas souhaitable de perdre l'information de l'existence de ce type dans le jeu de données. De même, les entités précédemment entrées ne doivent pas être considérées comme du bruit lorsqu'une entité est supprimée, car des entités similaires pourraient être insérées ultérieurement. Pour ces raisons, nous ne mettons pas à jour les clusters lorsqu'une entité est supprimée. L'insertion d'une nouvelle entité peut provoquer la fusion de deux clusters, ce qui dans notre cas correspond à la perte de deux types, ce qui n'est pas souhaitable. Au lieu de fusionner les

deux clusters, nous allons assigner l'entité aux deux. Outre l'insertion ou la suppression d'entités, une autre évolution qui peut se produire est la modification de l'ensemble des propriétés qui décrivent une entité ; nous considérons que cela équivaut à une suppression suivie d'une insertion.

Pour trouver le(s) type(s) d'une nouvelle entité e, nous pouvons utiliser un algorithme de classification, tel que *K-NN* (K-Nearest Neighbors) [81], qui est un algorithme de raisonnement à partir de cas. Il prend des décisions en recherchant un ou plusieurs cas similaires déjà résolus. L'idée générale est de trouver pour un nouvel objet, les k objets les plus similaires appartenant à l'ensemble d'apprentissage. Cependant, cet algorithme nécessite de fournir en paramètre le nombre de voisins les plus proches k. Pour pouvoir assigner plusieurs types à une entité, il est également nécessaire de spécifier le nombre de types que va avoir une entité ; or, nous ne pouvons pas prédire *a priori* le nombre de types que peut avoir une entité. De plus, cet algorithme est coûteux car il recherche les voisins les plus proches dans tout le jeu de données.

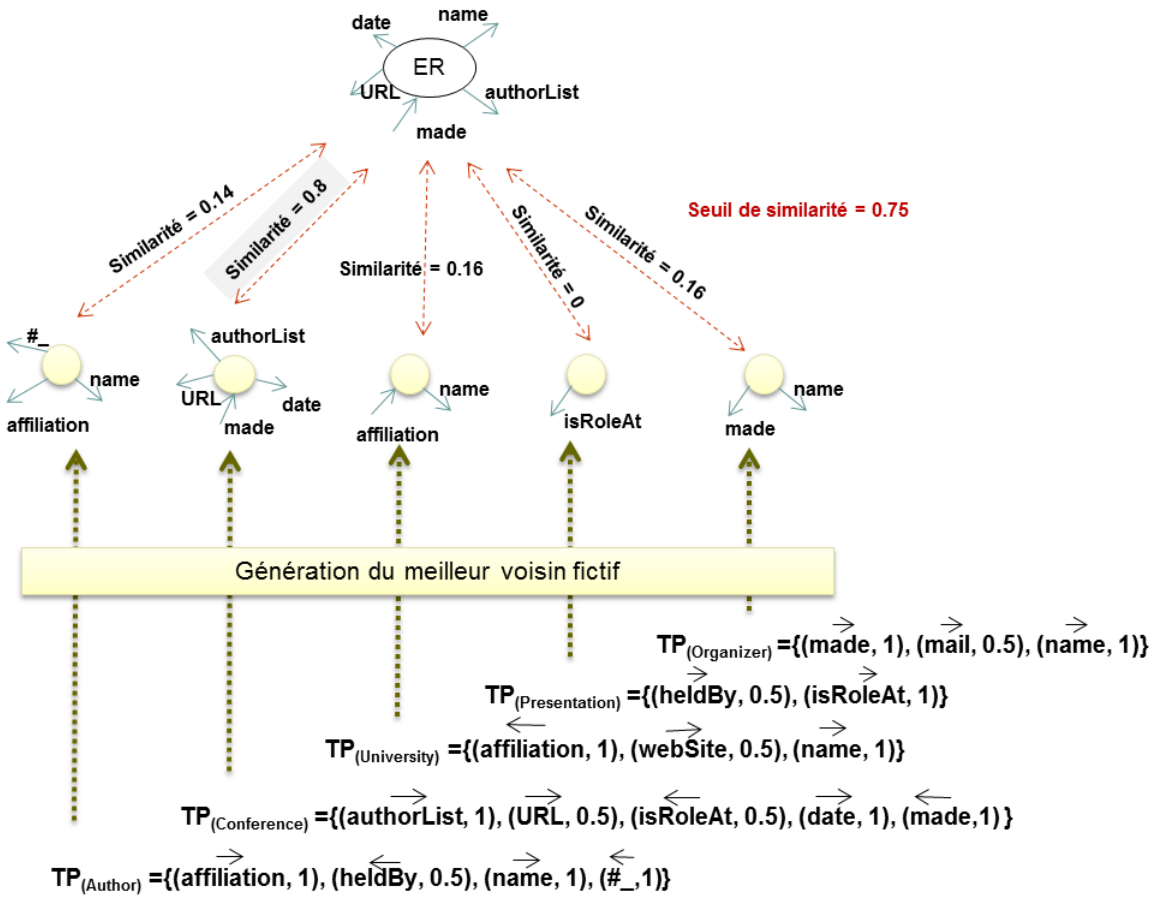


FIGURE 3.12 – Génération de voisins fictifs pour une nouvelle entité

Comme les données du Web sémantique sont très hétérogènes, il faut

maximiser les chances d'affectation d'une nouvelle entité aux types existants. Nous proposons pour cela une méthode originale de classification spécialement imaginée pour les données du Web sémantique. Pour classer une nouvelle entité e, nous proposons de générer un voisin fictif b appelé *meilleur voisin fictif* de e à partir de chaque profil de type TP dans le jeu de données. Le meilleur voisin fictif de e généré à partir de TP est composé des propriétés communes entre e et TP et des propriétés fortes de TP. Nous définissons le meilleur voisin fictif comme suit.

Définition (Meilleur voisin fictif). Le meilleur voisin fictif b d'une entité e en considérant un profil de type TP est une entité fictive décrite par $b.P_U$ qui est un ensemble de propriétés non primitives, tel que :
- Si $p \in e.P_U$ et $(p, \alpha) \in TP$, alors $p \in b.P_U$;
- Si $(p, 1) \in TP$, alors $p \in b.P_U$.

Nous calculons la similarité entre une nouvelle entité e et son meilleur voisin fictif b de la même manière qu'entre deux entités au cours du processus de regroupement. Cela en adaptant la similarité de Jaccard pour mesurer la similarité entre leurs deux ensembles de propriétés respectifs $e.P_U$ et $b.P_U$, comme suit :

$$Similarité(e, b) = \frac{|e.P_U \cap b.P_U|}{|e.P_U \cup b.P_U|} \tag{3.3}$$

Nous considérons les deux entités b et e comme des voisins si la similarité entre elles est supérieure au seuil de similarité ϵ qui a été déterminé dans la section 3.5.3. Dans ce cas, nous assignons à e le type du profil d'où le meilleur voisin fictif b a été extrait. Ce processus est répété pour chaque profil de type dans la source de données.

La figure 3.12 montre un exemple d'ajout d'une nouvelle entité dans un jeu de données déjà typé. Le profil de chaque type existant dans le jeu de données caractérise la structure des entités de ce type. Pour trouver le(s) type(s) de la nouvelle entité e, le meilleur voisin fictif de cette entité est extrait de chaque profil de type. Le meilleur voisin fictif de l'entité e par rapport à un profil de type TP est composé des propriétés communes entre l'entité e et le profil TP et les propriétés fortes de TP. Par exemple, le meilleur voisin fictif de l'entité e par rapport au profil du type *Organizer* est $b = \{\overrightarrow{name}, \overrightarrow{made}\}$. Dans notre exemple, la similarité entre chaque voisin fictif et la nouvelle entité e est calculée. En considérant un seuil de similarité de 0.75, seul le type *Conference* est attribué à la nouvelle entité e.

Lors de la recherche des types d'une nouvelle entité, trois cas sont possibles :
- Il y a des types découverts pour la nouvelle entité, alors ces types sont affectés aux entités bruitées similaires à la nouvelle entité ;
- Il n'y a pas de types découverts pour la nouvelle entité, et celle-ci est simi-

Algorithm 6: Typage incrémental pour une entité

Input : nouvelle entité e, profils de types $\{TP_{class}\}$, l'ensemble des entités bruitées N, seuil de similarité ϵ
Output: type(s) de e

```
1  A = e;
2  for each TP_class do
3  |  Extraire le meilleur voisin fictif b pour e à partir de TP_class;
4  |  if Similarité(b,e) >= ε then
5  |  |  Ajouter le type de TP_class à e;
6  |  end
7  end
8  for each o ∈ N do
9  |  if Similarité(o,e) >= ε then
10 |  |  grouper o et e dans A;
11 |  end
12 end
13 if e a des types assignés then
14 |  Ajouter les types de e à chaque entité o ∈ A;
15 |  if e a uniquement un type T_class then
16 |  |  for each r ∈ A do
17 |  |  |  Mettre à jour le profil de TP_class;
18 |  |  end
19 |  end
20 else
21 |  if |A| > 1 then
22 |  |  A représente un nouveau type;
23 |  |  Créer un nouveau profil pour le type A;
24 |  else
25 |  |  e est une entité bruitée;
26 |  end
27 end
```

laire à des entités bruités, alors un nouveau type est construit, regroupant la nouvelle entité et ses entités bruitées similaires ;
- Il n'y a pas de types découverts pour la nouvelle entité, et celle-ci n'est similaire à aucune entité bruitée, alors la nouvelle entité est considérée comme bruitée.

Nous ne mettons à jour un profil de type que si l'entité e a un type unique, car si nous considérons des entités ayant plusieurs types dans les profils de type, nous pourrions y introduire des propriétés caractérisant un autre type.

L'**algorithme** 6 permet de trouver les types d'une nouvelle entité. Il prend

en entrée : la nouvelle entité e, les profils des types existants dans le jeu de données, l'ensemble des entités bruitées N et le seuil de similarité ϵ. Au départ, un cluster potentiel A est créé pour contenir la nouvelle entité e. Un voisin fictif de e est extrait de chaque profil de type. Si la similarité entre e et un voisin fictif b est supérieure au seuil de similarité ε, alors le type du profil, d'où a été extrait b, est ajouté comme type de e. Ensuite, la similarité entre e et chaque entité bruitée est calculée. Si la similarité entre e et une entité bruitée est supérieure au seuil de similarité ε, alors l'entité n'est plus considérée comme du bruit et elle est ajoutée au cluster potentiel A. Si e a des types assignés, alors il faut assigner ses types à toutes les entités contenues dans le cluster potentiel A. Si e a un type unique le profil correspondant est mis à jour à l'aide de la fonction **MettreAJourProfil**, qui est décrite dans l'**algorithme 3**. Cette fonction permet d'ajouter les propriétés qui ne sont pas déjà dans le profil et elle met à jour les probabilités des propriétés existantes. Cette fonction est également utilisée pour créer les profils des types pendant le processus de clustering. Si l'entité e n'a aucun type assigné et si son cluster potentiel A contient plus d'une entité, alors A est considéré comme une nouveau type, et son profil est construit. En revanche, si le cluster potentiel A ne contient que l'entité e, alors celle-ci est considérée comme du bruit.

L'**algorithme** 6 a une complexité de $O(k)$, où k représente le nombre de types dans le jeu de données.

3.6 Génération des liens

Un schéma extrait est composé des types contenus dans la source de données et des liens entre ces types. En effet, en plus de la découverte de types, les liens entre ces types sont également importants pour comprendre le contenu d'une source de données.

Afin d'obtenir le schéma complet d'une source de données liées, nous proposons une approche pour générer les liens entre les types découverts. Cela, en analysant les liens qui existent entre les entités. Nous nous sommes intéressés à deux types de liens : (i) les liens sémantiques, correspondants à des propriétés non primitives et (ii) les liens hiérarchiques, correspondants à la propriété $rdfs{:}subClassOf$. Nous présentons notre approche de découverte de liens sémantiques dans la section 3.6.1, et nous présentons celle de la découverte des liens hiérarchiques dans la section 3.6.2.

3.6.1 Liens sémantiques

Un lien sémantique est généré sur la base d'une propriété définie par l'utilisateur et qui lie des entités. Un lien sémantique généré sur la base d'une propriété p entre deux types T_i et T_j signifie que le domaine de la propriété p est le type

T_i et son co-domaine est le type T_j ce qui est respectivement pour les triplets (p $rdfs{:}domain$ T_i) et (p $rdfs{:}range$ T_j).

FIGURE 3.13 – Les liens sémantiques découverts entre les types du jeu de données de la figure 3.1

Dans les travaux présentés dans l'état de l'art, la problématique de la découverte des liens sémantiques entre les types des données n'est pas spécifiquement abordée. Néanmoins, l'approche présentée dans [26] propose d'extraire le schéma d'un jeu de données liées en inférant également les liens sémantiques entre les types découverts, et cela en parcourant les entités du jeu de données, et à chaque fois qu'une entité est liée à une autre par une propriété, un lien pour cette propriété est généré entre les types de ces entités. Cependant, cette approche nécessite de parcourir tout le jeu de données ce qui peut être très coûteux.

Nous proposons de générer des liens sémantiques entre les types découverts en analysant leurs profils. Si une propriété sortante dans un profil de type, est aussi entrante dans un autre profil de type, alors un lien sémantique est possible entre ces deux types. Ce lien sémantique est validé s'il existe deux entités appartenant respectivement à chacun des types qui sont liés par cette propriété. La figure 3.13 montre les liens sémantiques générés entre les types découverts du jeu de données de la figure 3.1. Par exemple, les types *Organizer* et *Conference* sont liés par le lien sémantique *made*, car dans le profil du type *Organizer*, il y la propriété sortante $\overrightarrow{made}$, et dans le profil du type *Conference*, il y la propriété entrante $\overleftarrow{made}$. Ce lien est validé par le fait qu'une entité de type *Organizer* liée à une entité de type *Conference* par la propriété *made*, comme pour les entités *P4* et *ISWC* de la figure 3.1. Un lien sémantique est défini comme suit.

Définition (Lien sémantique). Deux types T_i, T_j sont liés par une propriété p si $\overrightarrow{p}$ appartient à l'ensemble des propriétés du profil de type TP_i et $\overleftarrow{p}$ appartient à l'ensemble des propriétés du profil de type TP_j, comme suit :

– Si ($\exists\ p : (\overrightarrow{p}, \alpha_i)$ dans $TP_i \wedge (\overleftarrow{p}, \alpha_j)$ dans TP_j) alors (il peut exister un lien

sémantique $\overrightarrow{p}$ de T_i vers T_j).
- Le lien $\overrightarrow{p}$ est validé en trouvant deux entités $e \in T_i$ et $e' \in T_j$ tel que $(e, p, e') \in D$.

Nous proposons également de pondérer chaque lien par deux poids, l'un sortant et l'autre entrant, pour refléter la proportion des entités de chaque type concerné par un lien donné. Soit p un lien entre deux types T_i et T_j. Le poids sortant de p représente le rapport entre le nombre d'entités de type T_i liées par $\overrightarrow{p}$ à des entités de type T_j et le nombre d'entités de type T_i. Le poids entant de p représente le rapport entre le nombre d'entités de type T_j liées par $\overleftarrow{p}$ à des entités de type T_i et le nombre d'entités de type T_j.

3.6.2 Liens hiérarchiques

Un lien hiérarchique est généré s'il existe des types qui peuvent être regroupés en un type plus générique. Ces types seront liés à ce type générique par la propriété $rdfs{:}subClassOf$, comme les types *Organizer* et *Author* qui sont liés au type générique *Person* dans la figure 3.14.

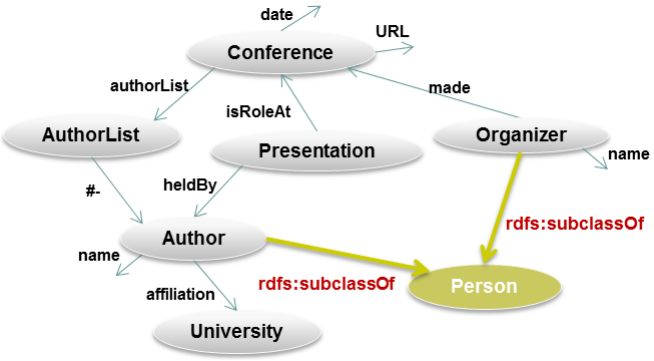


FIGURE 3.14 – Les liens hiérarchiques découverts entre les types du jeu de données de la figure 3.1

La problématique de la découverte des liens hiérarchiques a été plus ou moins abordée par les approches existantes de découverte de schéma. L'approche proposée dans [26] découvre le schéma d'un jeu de données liées en appliquant un algorithme de clustering hiérarchique. Par conséquent, des liens hiérarchiques sont automatiquement générés entre les données. L'approche proposée dans [43] utilise également un clustering hiérarchique, mais pour construire une hiérarchie des types déclarés dans le jeu de données. Les déclarations $rdf{:}type$ des instances sont exploitées à cette fin. Cependant, un algorithme de clustering hiérarchique appliqué au jeu de données pour générer ces liens est très coûteux. Pour trouver la meilleure partition, toute la hiérarchie générée doit être explorée. De plus,

de nombreux liens hiérarchiques entre les données sont obtenus à l'issue de ce processus, sans pour autant être tous significatifs.

Nous proposons de découvrir des liens hiérarchiques en générant une hiérarchie à partir des profils de types. Ce qui est moins coûteux que l'application d'un algorithme de clustering hiérarchique sur tout le jeu de données car le nombre de profils est généralement très faible par rapport à la taille du jeu de données.

Algorithm 7: Génération des liens hiérarchiques

Input : $ensProfilsType$
Output: $ensLiensHirarchiques$

```
ensLiensHirarchiques ← ∅;
while |ensProfilsType| > 1 do
    Trouver les profils de types les plus similaires et soit la valeur de cette
     similarité meilleurSimilarit;
    if meilleurSimilarit = 0 then
        Grouper tout le reste des profils de type dans le type générique
         Thing et Arrêter;
    else
        Construire le profil du type générique qui regroupe les profils de
         type les plus similaires;
        ensLiensHirarchiques ← ensLiensHirarchiques ∪
         {rdfs:subClassOf liens entre ces plus similaires profils de type et
         le type générique};
        ensProfilsType ← ensProfilsType ∪ {le profil du type
         générique} ;
        Supprimer ces profils de types les plus similaires de
         ensProfilsType ;
    end
end
```

Notre procédure de découverte de liens hiérarchiques est décrite dans l'**algorithme 7**. Nous avons adapté un algorithme de clustering hiérarchique ascendant ; le profil du type générique est construit à chaque étape de la hiérarchie. Cet algorithme prend en entrée l'ensemble des profils des types contenus dans le jeu de données. Il renvoie l'ensemble des liens hiérarchiques entre les types. A chaque itération, les profils de types les plus similaires sont regroupés pour former un profil de type générique qui les remplacera dans le processus. Un lien de hiérarchie est généré entre chaque type et leur type générique. L'algorithme itère tant que tous les types n'ont pas été regroupés en un seul type générique *Thing*, ou tant que la meilleure valeur de similarité entre les profils des types restants n'est pas égale à 0. Le type générique de deux types est défini comme suit.

Définition (Type générique). Un type générique T_g entre deux types T_i et T_j, est formé sur la base de la similarité de ces deux types. Le profil du type générique TP_g est composé de toutes les propriétés des profils respectifs de TP_i et TP_j. La probabilité d'une propriété est recalculée comme probabilité moyenne pondérée par la cardinalité de chaque type, comme suit :

– $\forall (p, \alpha) \in (TP_i \cup TP_j) : (p, Prob_{i,j}(p)) \in TP_g$

avec :

$$Prob_{i,j}(p) = \frac{\alpha_i \times |T_i| + \alpha_j \times |T_j|}{|T_i| + |T_j|} \tag{3.4}$$

Pour calculer la similarité entre deux profils de types, nous définissons une nouvelle mesure de similarité inspirée de la mesure de similarité de Jaccard, comme suit.

Définition (Similarité entre deux profils de types). La similarité entre deux profils de types TP_i et TP_j, est calculée comme le rapport entre le nombre de propriétés en commun pondérées par leurs probabilités pour chaque type et la cardinalité de l'union des propriétés pondérées par leurs probabilités dans un type générique potentiel (voir formule 3.4). Cette mesure est définie comme suit :

$$ProfileSim(TP_i, TP_j) = \frac{\sum_{\forall p \in \{TP_i \cap TP_j\}} Prob_{i,j}(p)}{\sum_{\forall p \in \{TP_i \cup TP_j\}} Prob_{i,j}(p)} \tag{3.5}$$

Cette approche génère une hiérarchie en regroupant les types en paires afin de trouver leur type générique. Toutefois, certains types génériques peuvent être composés de plus de deux sous-types comme dans la figure 3.15 (a). Dans ce cas, on peut détecter facilement les types génériques intermédiaires redondants et les éliminer (comme dans la figure 3.15 (b)), puisque le nombre de types dans un jeu de données n'est généralement pas élevé.

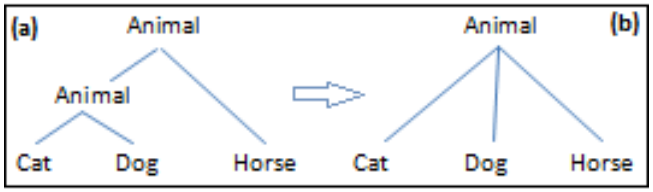


FIGURE 3.15 – Élimination des types génériques redondant dans une hiérarchie.

3.7 Évaluation

Cette section présente quelques résultats d'expérimentation de nos approches. Dans un premier temps, nous avons généré le seuil de similarité, et nous avons évalué la qualité du schéma généré pour différentes sources de données. Puis, nous avons évalué le seuil de similarité proposé compte tenu du nombre de clusters générés, du pourcentage de bruit et de la qualité des types générés.

3.7.1 Jeux de données

Nous avons évalué nos approches sur trois jeux de données : le jeu de données *Conference*[7], qui décrit les données de plusieurs conférences et ateliers du Web sémantique avec 11 types et 1430 triplets ; le jeu de données *BNF*[8] qui contient des données sur la Bibliothèque Nationale de France avec 5 types et 381 triplets ; nous avons extrait un jeu de données de *DBpedia*[9] avec 19696 triplets en considérant les types suivants : *Politician*, *SoccerPlayer*, *Museum*, *Movie*, *Book* et *Country*.

Les tests ont été effectués sur une machine Intel (R) Xeon (R), CPU de 2.80 GHz, 64 bits avec 4 Go de RAM.

3.7.2 Métriques et méthodologie expérimentale

Nous avons généré le seuil de similarité, et nous avons évalué la qualité du schéma généré pour différentes sources de données. Afin d'évaluer la qualité des résultats obtenus par nos algorithmes, nous avons extrait les déclarations de types existantes (*rdf:type*) dans nos sources de données et nous les avons considérées comme une référence. Nous avons ensuite exécuté notre algorithme de découverte de types sur les jeux de données sans les déclarations de types et nous avons évalué la précision et le rappel pour les types découverts. Nous avons annoté chaque cluster de type découverts C_i avec l'étiquette du type le plus fréquent associé à ses entités. Pour chaque étiquette de type L_i correspondante au type déclaré T_i dans le jeu de données et chaque type découvert C_i déduit par notre algorithme, tel que L_i est l'étiquette de C_i, nous avons évalué la précision $P_i(T_i, C_i) = |T_i \cap C_i|/|C_i|$ et le rappel $R_i(T_i, C_i) = |T_i \cap C_i|/|T_i|$. Nous avons fixé $MinPts$ à 1 de sorte qu'une entité est considérée comme un bruit si elle n'a pas de voisins, comme expliqué dans la section 3.4.1.

Pour évaluer la qualité globale des types et des liens sémantiques et hiérarchiques découverts, nous avons utilisé les mesures de précision et de rappel. Soient k le nombre de types découverts, et n le nombre d'entités dans le jeu de données.

7. *Conference* : data.semanticweb.org/dumps/Conferences/dc-2010-complete.rdf
8. *BNF* : datahub.io/fr/jeu de données/data-bnf-fr
9. *DBpedia* : dbpedia.org

Pour évaluer la qualité globale des types découverts, chaque type est pondéré en fonction de son nombre d'entités comme suit :

$$P = \sum_{i=1}^{k} \frac{|C_i|}{n} \times P_i(T_i, C_i) \tag{3.6}$$

$$R = \sum_{i=1}^{k} \frac{|C_i|}{n} \times R_i(T_i, C_i) \tag{3.7}$$

La précision et le rappel des liens générés sont évalués en tenant compte des vrais/faux positifs et des faux négatifs. Nous avons comparé le schéma inféré à celui du jeu de données s'il est fourni, comme pour le jeu de données *BNF*. Si aucun schéma n'est fourni, nous l'avons modélisé manuellement en nous basant sur les informations sur le schéma fournies dans le jeu de données ; à cette fin, nous avons construit l'ensemble des propriétés primitives D_P tel que décrit dans la section 3.2.1.

Pour montrer que notre approche peut être utilisée pour trouver le type des nouvelles entités entrantes dans le jeu de données, nous avons utilisé une technique de validation croisée en partitionnant aléatoirement chaque jeu de données en un ensemble d'apprentissage (2/3 du jeu de données) et un ensemble de test (1/3 du jeu de données). Nous avons appliqué notre approche auto-adaptative sur l'ensemble d'apprentissage pour construire les profils de type. Nous avons appliqué notre approche de typage incrémental sur les entités de l'ensemble test pour trouver les types appropriés pour chaque entité. La précision de l'approche est le pourcentage d'entités correctement classées dans l'ensemble de test en fonction de leurs types déclarés dans le jeu de données. Le rappel de l'approche est le pourcentage d'entités d'un type donné qui ne sont pas trouvées dans l'ensemble de test. Soient TP le nombre de vrais positifs, FP le nombre de faux positifs et FN le nombre de faux négatifs, nous évaluons la précision P, le rappel R et la mesure $F1$ comme suit :

$$P = \frac{TP}{TP + FP} \tag{3.8}$$

$$R = \frac{TP}{TP + FN} \tag{3.9}$$

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{3.10}$$

Pour évaluer le seuil de similarité proposé par notre algorithme, nous avons fait varier la valeur du seuil de similarité et nous avons comparé : (i) le nombre de clusters résultants avec le nombre de types déclarés dans le jeu de données, (ii) le pourcentage de bruit détecté et (iii) la qualité des types découverts. Pour évaluer

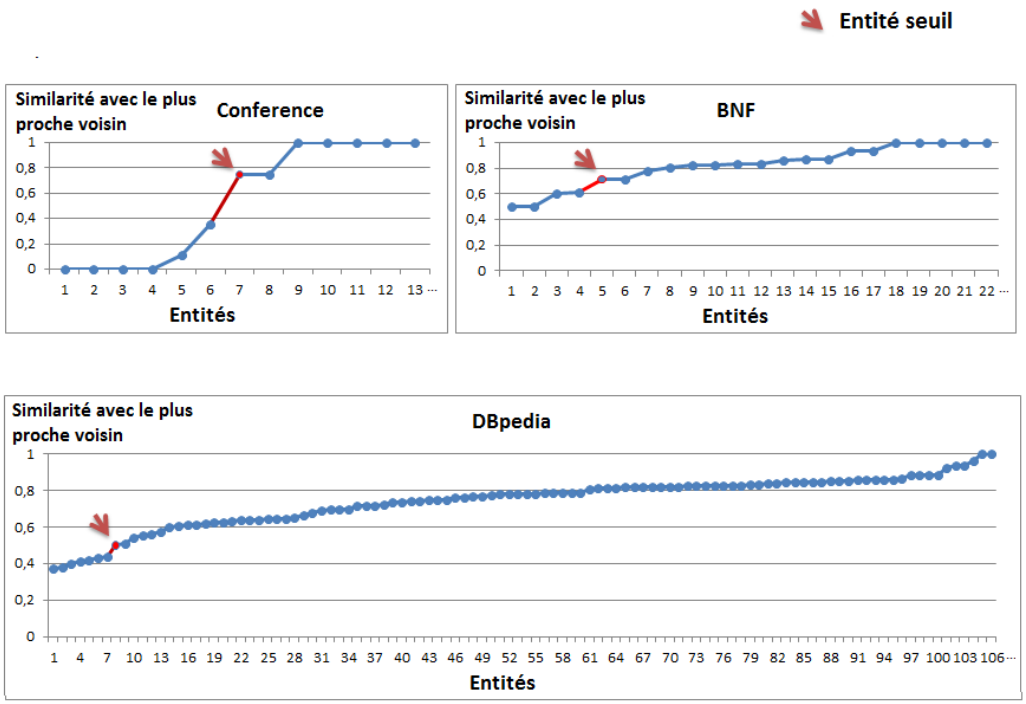


FIGURE 3.16 – Détection automatique du seuil de similarité.

la qualité des types découverts, nous avons exécuté nos algorithmes sur les jeux de données sans les déclarations de types, avec différents seuils de similarité, puis nous avons évalué la précision et le rappel pour les types découverts comme décrit dans les formules 3.6 et 3.7.

3.7.3 Résultats

Nous avons automatiquement détecté le seuil de similarité ε pour chaque jeu de données. La figure 3.16 représente les entités de chaque jeu de données ordonnées en fonction des valeurs décroissantes de leurs valeurs de similarité avec leurs plus proches voisins. Nous n'avons pas représenté toutes les entités du jeu de données, car la valeur de similarité avec le voisin le plus proche reste la même quand elle atteint 1. On considère l'entité ayant le plus grand écart avec la similarité de l'entité qui la précède comme l'*entité seuil*. Les entités à gauche de cette *entité seuil*, dans la figure 3.16, sont considérées comme du bruit. La valeur de similarité de l'*entité seuil* avec son plus proche voisin, représente la valeur de ε de chaque jeu de données ; les résultats obtenus sont : $\varepsilon = 0.75$ pour le jeu de données *Conference* ; $\varepsilon = 0.72$ pour le jeu de données *BNF* et $\varepsilon = 0.5$ pour le jeu de données de *DBpedia*.

Les entités de même type, dans les jeux de données *BNF* et de *Conference*,

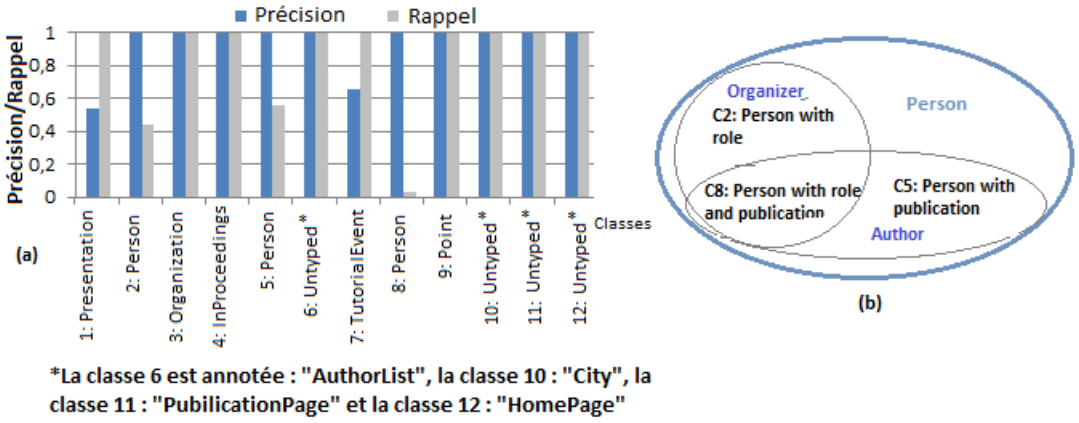


FIGURE 3.17 – Qualité des types générés (a) et les types qui se recouvrent (b) dans la source de données *Conference*.

ont des ensembles de propriétés plus homogènes que les entités dans *DBpedia*, ce qui explique pourquoi le seuil de similarité détecté est plus faible pour *DBpedia*. Le calcul de la similarité entre chaque paire d'entités a une complexité de $O(n^2)$, et le temps d'exécution pour chaque jeu de données est : de 16 millisecondes (ms) pour le jeu de données de *Conference* ; de 2 ms pour le jeu de données *BNF* ; et de 32 ms pour le jeu de données de *DBpedia*. La détection automatique du seuil de similarité prend quelques millisecondes pour chaque jeu de données car elle a une complexité linéaire $O(n)$ puisque la similarité entre une entité et son plus proche voisin est calculée au préalable.

La qualité de chaque type découvert dans la source de données *Conference* est illustrée par la figure 3.17 (a). Notre approche donne de bonnes valeurs de précision et de rappel ; de plus elle permet de découvrir des types non déclarés dans la source de données comme les types découverts 6, 10, 11 et 12 qui sont étiquetés respectivement *AuthorList*, *PublicationPage*, *HomePage* et *City*.

Dans certains cas, les types ont été découverts uniquement à partir de leurs propriétés entrantes. En effet, pour les conteneurs, tels que *AuthorList*, il est nécessaire de tenir compte de ces propriétés, car ils ne disposent pas de propriétés sortantes. Les types découverts 1 et 7 n'ont pas une bonne précision, car ils contiennent des entités de différents types dans le jeu de données. Cependant, ces types ont la même structure, il est donc impossible de les distinguer par un regroupement selon la similarité structurelle. Le rappel pour le type déclaré *Person* est faible, cela est dû au fait que les entités de ce type sont divisées en trois types découverts : le type découvert 8 représente les personnes qui ont publié et joué un rôle dans la conférence ; le type découvert 2 représente les personnes qui ont seulement joué un rôle (par exemple, président, membre du comité) ; le type découvert 5 représente les personnes qui ont seulement publié. Les types qui

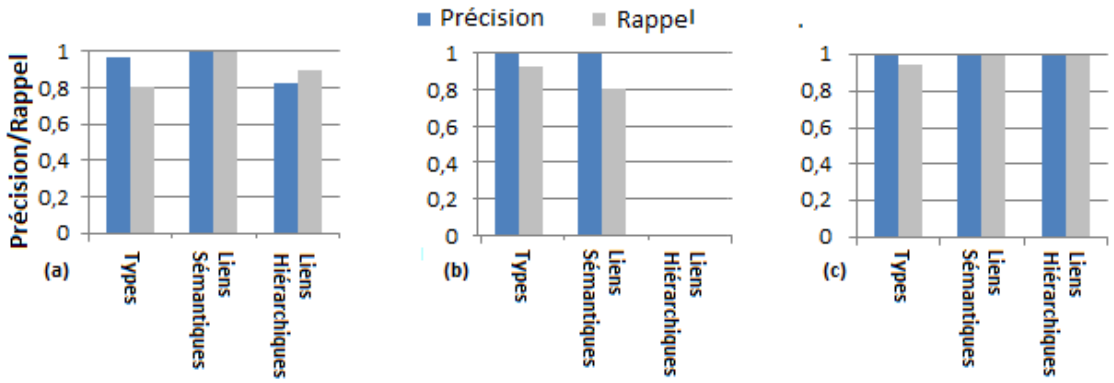


FIGURE 3.18 – Évaluation de la qualité du schéma découvert pour les jeux de données : *Conference* (a), *BNF* (b) et *DBpedia* (c).

se recouvrent sont découverts grâce à l'analyse des profils de type. Cela a conduit à des résultats présentés dans la figure 3.17 (b). Le type découvert 8 est associée à deux types déclarés : le premier du type découvert 2 (étiqueté manuellement *Organizer*) et le second du type découvert 5 (étiqueté manuellement *Author*), ce qui est conforme aux déclarations des entités du jeu de données. Notez que trouver les étiquettes des types découverts est un problème ouvert que nous allons aborder dans le chapitre qui suit.

Nous pouvons voir dans la figure 3.18 que l'approche donne une bonne précision et un bon rappel pour les schémas générés, composés de types, de liens sémantiques et hiérarchiques. Pour le jeu de données *Conference* (voir la figure 3.18 (a)), la précision n'est pas maximale en raison des types découverts 1 et 7 comme discuté précédemment. Le rappel n'est pas maximal car les entités de type *Person* sont divisées en trois comme discuté ci-dessus. Les résultats pour les jeux de données *BNF* (voir figure 3.18 (b)) et *DBpedia* (voir figure 3.18 (c)) montrent que la qualité de la génération du type des entités a une bonne précision et un bon rappel. Le rappel n'est pas maximal car des instances bruitées ont été détectées. Pour le jeu de données *BNF*, certains des liens sémantiques n'ont pas été déclarés dans le schéma fourni, ce qui explique pourquoi le rappel n'est pas maximal. Cependant, après avoir vérifié les entités du jeu de données, nous avons découvert que ces liens sémantiques étaient valides. Pour le jeu de données *DBpedia*, notre algorithme a pu différencier entre les entités des deux types *Politician* et *SoccerPlayer* même si elles ont des ensembles de propriétés similaires, comme le montrent les profils de types correspondants générés par notre algorithme :

- Politician : $\{(\vec{name}, 1), (\vec{party}, 0.73), (\vec{children}, 0.21), (\vec{birthDate}, 0.94), (\vec{nationality}, 0.15), (\overleftarrow{successor}, 0.78), (\vec{deathDate}, 0.68), ...\}$.
- SoccerPlayer : $\{(\vec{name}, 1), (\vec{height}, 0.46), (\vec{surname}, 0.93), (\vec{birthDate}, 1), (\vec{nationalteam}, 0.86), (\overleftarrow{currentMember}, 0.8), (\vec{deathDate}, 0.06), ...\}$.

Les profils de types générés reflètent l'hétérogénéité des données, par exemple, 6 % des entités de type *SoccerPlayer* ont la propriété sortante *deathDate*, et pourtant le regroupement généré était correct. Les résultats obtenus par notre approche sont bons même lorsque le jeu de données est composé d'entités de même type et décrites par des ensembles de propriétés hétérogènes.

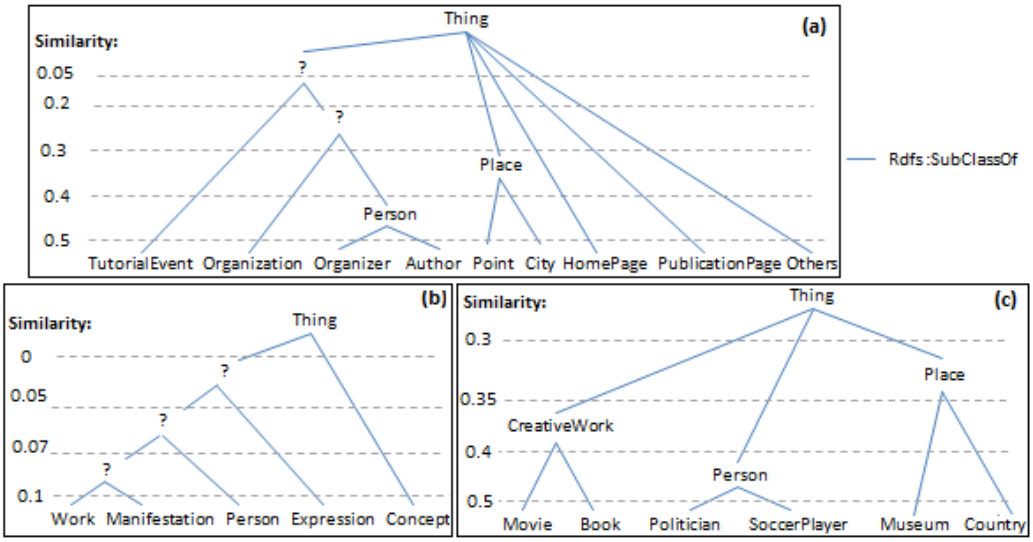


FIGURE 3.19 – Les liens hiérarchiques générés pour les jeux de données : *Conference* (a), *BNF* (b) et *DBpedia* (c).

Les liens hiérarchiques générés pour le jeu de données *DBpedia* sont corrects car ils sont conformes aux déclarations $rdfs{:}subClassOf$ existantes (voir figure 3.19 (c)). Le jeu de données *BNF* n'a pas de liens hiérarchiques, ce qui explique que les valeurs de précision et de rappel soient à 0 dans la figure. Lorsque la similarité entre deux profils est faible, la sémantique du type générique n'est pas claire. Ceci est représenté par un point d'interrogation dans la figure 3.19 (b). Il en va de même pour certains des liens hiérarchiques générés pour le jeu de données *Conference* (voir figure 3.19 (a)) : un type générique a été généré pour *Person* et *Organisation*, cependant, la similarité entre leurs profils de type est faible. Notre algorithme n'a pas pu identifier un type générique pour *HomePage* et *PublicationPage* parce que leurs profils de type ne partagent aucune propriété, ce qui explique le rappel faible.

La figure 3.20 montre la précision et le rappel du typage des nouvelles entités qui forment l'ensemble de test de *Conference*. Nous pouvons voir que notre approche obtient une bonne qualité des types attribués aux nouvelles entités. Les types *Presentation* et *TutorialEvent* n'ont pas une bonne précision car ils contiennent des entités ayant différents types dans le jeu de données. Cependant, ces types ont la même structure et il est donc impossible de les distinguer. Notre algorithme a fait la distinction entre les types *City* et *Point* bien qu'ils soient

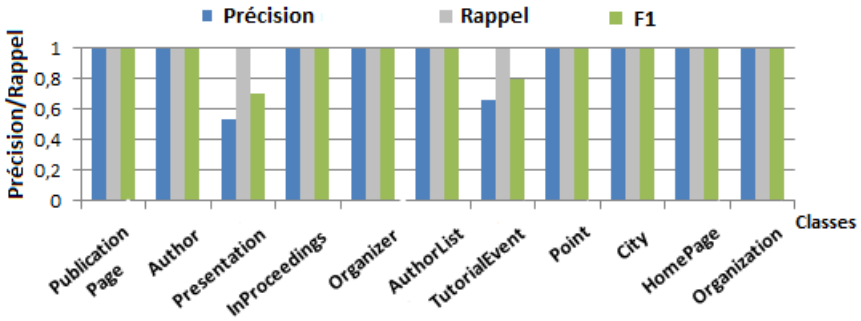


FIGURE 3.20 – La qualité des types attribués aux nouvelles entités du jeu de données *Conference*.

très semblables, comme ce fut le cas pour les types *Author* et *Organizer*. Pour les entités ayant à la fois les types *Author* et *Organizer*, l'approche a bien permis de trouver leurs types.

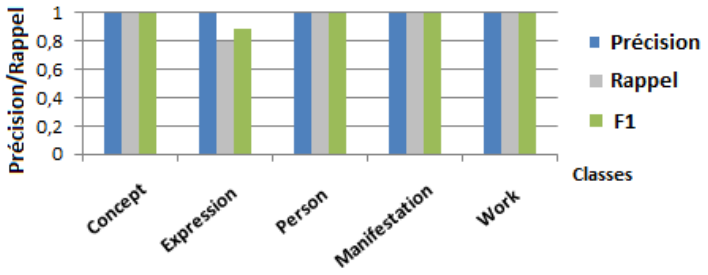


FIGURE 3.21 – La qualité des types attribués aux nouvelles entités du jeu de données *BNF*.

La figure 3.21 montre la précision et le rappel du typage des nouvelles entités qui forment l'ensemble de test de *BNF*. Nous pouvons voir que notre approche obtient de bonnes valeurs de précision et de rappel pour les entités nouvellement typées. Le type *Expression* n'a pas une bonne précision parce que certaines de ses entités sont considérées comme du bruit. Ces entités ont des déclarations *rdf:type Expression* dans le jeu de données, mais elles sont décrites par des ensembles de propriétés très différents des autres entités de ce type, donc elles sont considérées comme des entités bruitées par notre approche. Les types du jeu de données *BNF* sont assez dissimilaires, ce qui signifie que leurs profils ne partagent pas beaucoup de propriétés. Cela explique les valeurs élevées de la précision et du rappel de la classification.

La figure 3.22 montre la précision et le rappel du typage des nouvelles entités qui forment l'ensemble de test de *DBpedia*. Nous pouvons voir que notre approche donne des résultats de bonne qualité pour les entités classées. Cependant, les résultats ne sont pas aussi bons que pour les jeux de données *BNF* et

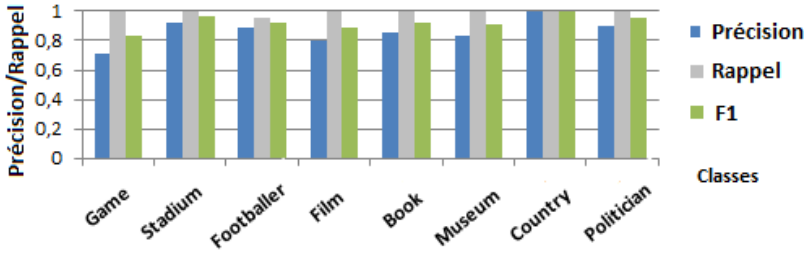


FIGURE 3.22 – La qualité des types attribués aux nouvelles entités du jeu de données de *DBpedia*.

de *Conference*, car il existe de nombreuses propriétés partagées entre les profils de types. En outre, le nombre moyen de propriétés d'une entité est de *150* dans *DBpedia*. Ces entités ont des propriétés générales qui ne sont pas spécifiques au type, telles que *wikiPageID*, *hasPhotoCollection*, *wikiPageDisambiguates*, *primaryTopic*, *hasPhotoCollection*, etc. Il existe certaines entités pour un type donné qui ne possèdent pas les propriétés spécifiques de ce type, par conséquent, elles sont détectées comme des instances bruitées par notre approche, comme pour les instances du type *Footballer*, qui n'ont pas de propriétés spécifiques de ce type, telles que *team* ou *club*.

Le processus de classification pour chaque jeu de données prend quelques millisecondes. Il a une complexité linéaire de $O(k)$, où k représente le nombre de types dans chaque jeu de données. Le nombre de types dans chaque jeu de données est généralement très faible par rapport au nombre d'entités. Le nombre de types est respectivement 11, 5 et 8 pour les jeux de données *Conference*, *BNF* et *DBpedia*.

La figure 3.23 indique le nombre de clusters (a), le pourcentage de bruit (b) et les précision/rappel des types générés en fonction du seuil de similarité dans le jeu de données *Conference*. Les résultats pour les jeux de données *BNF* et *DBpedia* sont donnés dans les figures 3.24 et 3.25 respectivement. La ligne verticale en pointillé représente le seuil de similarité proposé par notre approche. Pour le jeu de données *Conference* (voir figure 3.23), le nombre de clusters augmente lorsque le seuil de similarité augmente. En effet, plus le seuil de similarité est élevé, plus les entités tendent à être réparties dans des clusters différents. Le seuil de similarité déterminé par notre approche donne un nombre correct de clusters qui est égal au nombre de types du jeu de données. Le jeu de données *Conference* contient un très faible pourcentage de bruit (2.4 %) et la précision et le rappel sont tous deux assez élevés, en particulier avec le seuil de similarité généré. Le pourcentage de bruit reste stable même avec une valeur très élevée du seuil de similarité (0.95), car les entités du même type dans le jeu de données *Conference* ont un ensemble de propriétés très homogène.

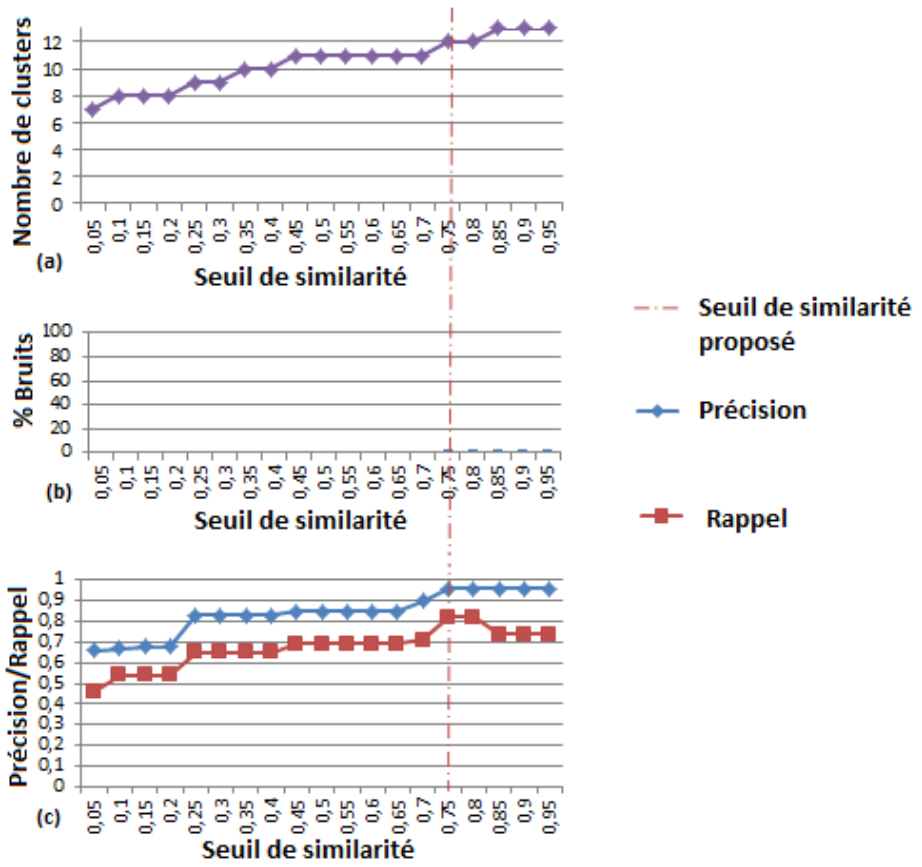


FIGURE 3.23 – Le nombre de clusters (a), le pourcentage de bruit (b) et les précision/rappel des types générés en fonction du seuil de similarité dans le jeu de données *Conference*.

Pour le jeu de données *BNF* (voir figure 3.24), le seuil de similarité proposé par notre approche donne un nombre correct de clusters qui est égal au nombre de types du jeu de données. La précision et le rappel varient en fonction du seuil de similarité. La précision augmente lorsque le seuil de similarité augmente. Une précision à *1* est obtenue avec le seuil de similarité proposé par notre approche. Cependant, le rappel diminue puis augmente quand on augmente le seuil de similarité, car si le seuil est élevé, les instances d'un type donné sont considérées comme du bruit alors qu'elles ne le sont pas. En effet, le pourcentage de bruit détecté augmente lorsque l'on augmente le seuil de similarité jusqu'à atteindre un pourcentage de 50 %, et dans ce cas, le rappel diminue car les instances typées dans le jeu de données ne sont pas regroupées en clusters à cause d'un seuil de similarité trop élevé. Le seuil de similarité proposé par notre approche donne un bon rappel avec un pourcentage de bruit raisonnable. Le rappel n'est pas maximal car certaines entités sont considérées comme du bruit parce qu'elles

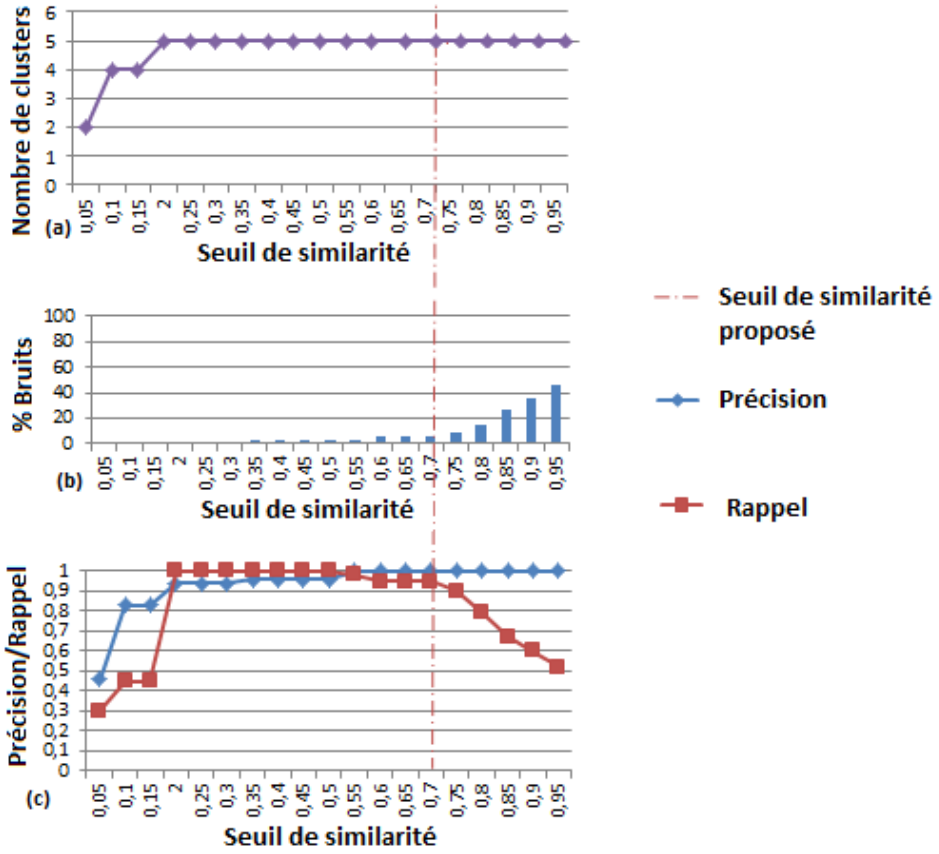


FIGURE 3.24 – Le nombre de clusters (a), le pourcentage de bruit (b) et les précision/rappel des types générés en fonction du seuil de similarité dans le jeu de données *BNF*.

sont très différentes des autres entités. C'est le cas pour certaines entités du type *Expression* qui sont considérées comme du bruit mais qui ont le type *Expression* dans le jeu de données.

Pour le jeu de données de *DBpedia* (voir figure 3.25), le nombre de clusters varie en fonction du seuil de similarité. En principe, plus le seuil de similarité est élevé, plus le nombre de clusters est élevé. Cependant, pour ce jeu de données, des instances qui ne sont pas bruitées sont considérées comme du bruit lorsque le seuil de similarité est trop élevé, ce qui a réduit le nombre de clusters. Le seuil de similarité proposé par notre approche donne un nombre correct de clusters qui est égal au nombre de types du jeu de données. La précision augmente à mesure que le seuil de similarité augmente pour atteindre une précision à *1*, qui est aussi la précision obtenue avec le seuil de similarité proposé par notre approche. Cependant, le rappel diminue puis augmente quand on augmente le seuil de similarité, parce que lorsque le seuil est élevé, les instances d'un type

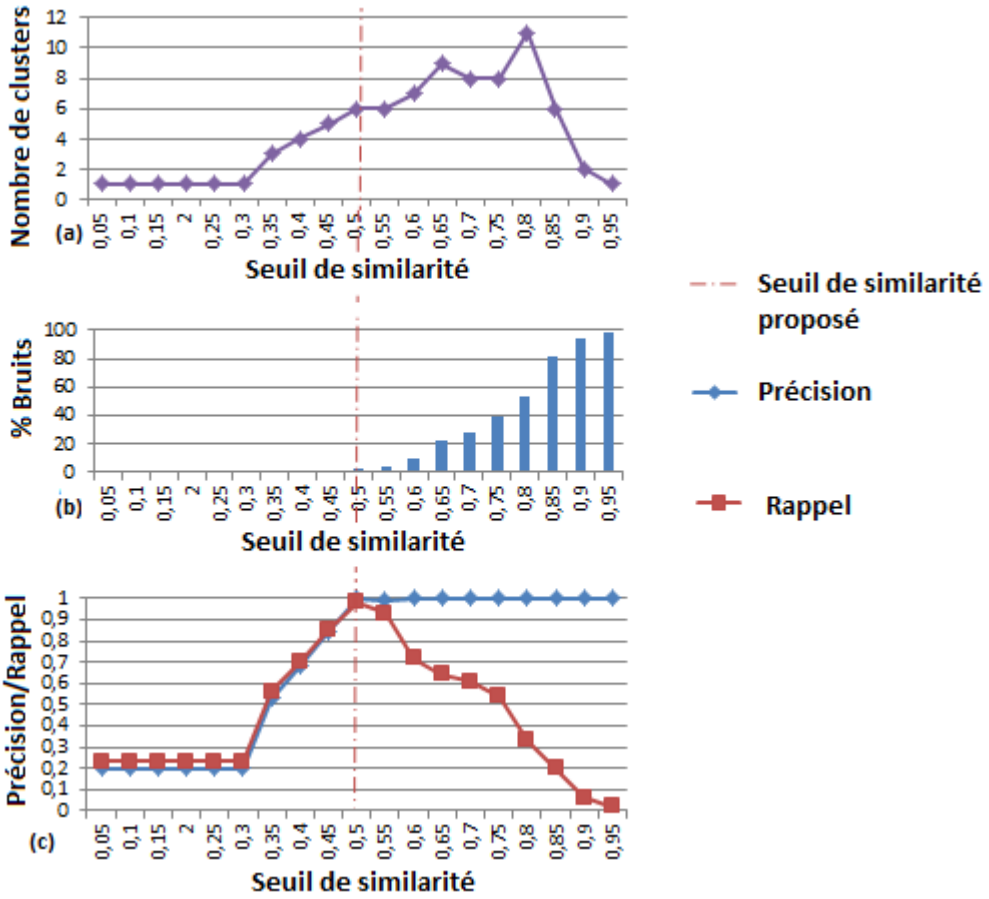


FIGURE 3.25 – Le nombre de clusters (a), le pourcentage de bruit (b) et les précision/rappel des types générés en fonction du seuil de similarité dans le jeu de données *DBpedia*.

donné peuvent être considérées comme du bruit alors qu'elles ne le sont pas. En effet, le pourcentage de bruit détecté augmente lorsque le seuil de similarité augmente jusqu'à atteindre un pourcentage de 98 %, et dans ce cas, le rappel diminue car certaines instances qui ont un type dans le jeu de données ne sont pas groupées dans un cluster, le seuil de similarité étant trop élevé. Le seuil proposé par notre approche donne un bon rappel avec un pourcentage de bruit raisonnable.

Les entités du même type dans le jeu de données *Conference* ont des ensembles de propriétés homogènes, donc, seul un faible bruit est détecté même lorsque le seuil de similarité est très élevé (0.95). Cependant, il existe des propriétés partagées entre les ensembles de propriétés d'entités de types différents, comme la propriété *name* entre les types *Organizer* et *Author* et la propriété *based-near* entre les types *City* et *Point*. Par conséquent, le nombre de types découverts (classes) varie selon le seuil de similarité plus que sa variation pour le jeu de

données *BNF*, où le nombre de types découverts reste le même et il est égal à 5 dès que le seuil de similarité atteint *0.2*. En effet, les entités de différents types dans le jeu de données *BNF* partagent très peu de propriétés. Cependant, les ensembles de propriétés des entités du même type sont moins homogènes que pour les données de *Conference*. Par conséquent, le bruit est plus important lorsque le seuil de similarité augmente. Les entités de *DBpedia* sont décrites par un grand nombre de propriétés (une moyenne de *150* propriétés par type). Certaines de ces propriétés ne sont pas spécifiques à un type, comme *hasPhotoCollection*, ce qui implique que les entités de différents types partagent de nombreuses propriétés. Par conséquent, le nombre de types découverts est de *1* lorsque le seuil de similarité est inférieur à *0.3*. De plus, les ensembles de propriétés des entités du même type sont très hétérogènes contrairement aux jeux de données *Conference* et *BNF*. Par conséquent, le seuil de similarité détecté est inférieur à celui détecté pour *Conference* et *BNF*. De plus, dans DBpedia l'augmentation du bruit est plus importante lorsque le seuil de similarité augmente. Le clustering de chaque jeu de données prend quelques secondes : étant donné n le nombre d'entités dans un jeu de données, la complexité du clustering est $O(n * log(n))$ et la similarité entre une entité et son plus proche voisin est calculée au préalable.

3.8 Conclusion

Obtenir une vue d'ensemble d'un grand graphe de données RDF et caractériser son contenu est souvent difficile. La compréhension d'une source de données devient encore plus difficile lorsque les informations relatives au schéma sont manquantes. Dans ce chapitre, nous avons proposé une approche auto-adaptative et incrémentale de découverte de schéma d'une source de données. Notre approche ne requiert pas de préciser le nombre de types dans la source de données, elle tient compte de l'évolution d'une source de données et elle peut affecter plusieurs types à une entité. Nous proposons une méthode auto-adaptative qui détecte automatiquement le seuil de similarité en fonction de la répartition de la similarité des entités avec leur voisin le plus proche. Nous proposons de construire un profil pour chaque type découvert au cours du processus de clustering pour résumer son contenu. Nous utilisons ces profils pour découvrir des clusters qui se recouvrent, ce qui permet d'attribuer plusieurs types à une entité. Pour pouvoir attribuer un type à une nouvelle entité sans parcourir le jeu de données, nous utilisons les profils en générant un *meilleur voisin fictif* pour la nouvelle entité. La description des types découverts est complétée par la génération de liens sémantiques et hiérarchiques entre eux. Nos expérimentations montrent que notre approche permet d'obtenir des résultats de bonne qualité en ce qui concerne les types et les liens dans le schéma découvert, même lorsque les entités sont très hétérogènes, comme pour *DBpedia*. Nos expérimentations montrent également que notre approche détecte un bon seuil de similarité sur différents jeux de données, en tenant compte

du nombre réel de types, du pourcentage de bruit et de la qualité des résultats en fonction de la précision et du rappel. En outre, l'approche donne également des résultats de bonne qualité pour le typage incrémental.

Notre approche permet de caractériser le contenu d'un jeu de données en fournissant un schéma comportant les types contenus dans ce jeu de données ainsi que les liens entre eux. Ceci est utile pour interroger une source de données, car il fournit les propriétés et les types existants dans cette source de données. Les profils pourraient également être utilisés pour l'interconnexion et la mise en correspondance des jeux de données. En effet, la mise en correspondance des types d'abord, lorsqu'ils sont connus, pourrait réduire considérablement la complexité du processus. Contrairement à certaines approches présentées dans l'état de l'art [8, 40, 36, 43], notre approche ne requiert aucune déclaration sur le schéma pour le découvrir, car elle utilise uniquement la structure implicite des données. Notre approche ne requiert aucun paramètre contrairement aux approches [31, 26] où le nombre de clusters ou le seuil de similarité sont à préciser. Notre approche permet de découvrir les recouvrements entre les types, ce qui permet d'attribuer plusieurs types à une entité, contrairement aux approches [4, 31, 26].

Une perspective intéressante pour notre travail est d'utiliser les technologies big data pour passer à l'échelle. Des adaptations fondées sur l'indexation et une représentation compacte des données pourraient être investiguées également pour optimiser l'utilisation de la mémoire et éviter le chargement des données. Une autre perspective importante est l'utilisation des profils lors du traitement d'une requête qui s'exécute sur des sources de données distribuées. Ces profils pourraient être utilisés pour décomposer la requête et envoyer les sous-requêtes aux sources pertinentes. Les probabilités des propriétés pourraient être utilisées pour optimiser le plan d'exécution en ordonnant les sous-requêtes en fonction de la sélectivité de leurs critères.

Dans notre approche, nous avons fait le choix de ne pas remettre en question les types découverts lors de l'évolution des données. Cependant, il serait intéressant d'investiguer le cas d'une évolution importante des données qui pourrait modifier les types existants. Nous avons vu dans l'état de l'art des approches relatives à l'extraction d'information sur le schéma qui utilisent les déclarations fournies dans le jeu de données pour les compléter et les enrichir. Une perspective intéressante, serait de voir s'il est possible d'envisager une approche hybride de découverte de schéma à partir de la structure implicite des données tout en exploitant les déclarations sur le schéma fournies dans le jeu de données. Une autre perspective est de proposer une approche qui permet de valider le schéma découvert. En effet, les déclarations sur le schéma fournies dans le jeu de données, pourraient être utilisées, par exemple : deux entités ayant la même valeur pour la déclaration *rdf:type* doivent appartenir au même type découvert.

Les données du Web sémantique sont très hétérogènes. Des entités de même type sont décrites par des ensembles de propriétés différents. Comme le profil

d'un type ne permet pas de renseigner la co-occurrence entre les propriétés mais seulement leur probabilités, alors nous proposons, dans le chapitre suivant, de décrire un type par ses versions possibles. Une version d'un type est un ensemble de propriétés qui décrivent une entité de ce type dans le jeu de données.

Un problème important qui est également abordé dans le chapitre 5 est l'annotation des types inférés. En effet, la découverte de groupes d'entités partageant le même type ne permet pas de connaître la sémantique du type, comme le fait qu'il s'agisse d'une université ou d'une conférence. Pour trouver la dénomination d'un type, nous proposons de capturer la sémantique des entités formant le groupe du type. Cela, en identifiant les annotations qui les décrivent au mieux à partir de sources de connaissances externes.

Chapitre 4

SchemaDecrypt : Découverte des versions des types d'une source de données distante sur le Web

4.1 Introduction

Dans le chapitre précédent, nous avons présenté une approche pour la découverte du schéma d'un jeu de données en terme de types et de liens entre eux. Dans notre approche de découverte de schéma, un type découvert est caractérisé par un profil probabiliste qui reflète la probabilité d'une propriété de décrire une instance du type. Par exemple, un profil de type peut refléter qu'une personne est décrite avec une adresse avec une probabilité de 0.5, et avec un e-mail avec une probabilité de 0.4. Cependant, cela ne spécifie pas qu'il existe des instances décrites par les deux propriétés adresse et e-mail en même temps. Ce profil de type n'est pas suffisant pour interroger une source de données car il ne permet pas de renseigner la co-occurrence entre les propriétés du type.

En plus des types et des liens entre eux dans un jeu de données, les versions d'un type permettent de caractériser la source de données avec un degré de précision plus important. Les différentes structures des instances d'un type représentent les différentes versions de ce type. Les versions de type permettent de décrire la co-occurrence entre les propriétés et leur nombre d'occurrences. Ce qui est utile par exemple lors de l'interrogation d'une source de données. Si une requête porte sur deux propriétés A et B et qu'aucune version du type n'a à la fois ces deux propriétés, alors on sait que la requête n'a pas de réponse dans cette source de données.

Dans l'état de l'art, nous avons présenté un ensemble d'approches de découverte de patterns structurels (versions des types). Ces approches ont été proposées dans différents contextes : (i) pour des sources de données locales ; (ii) pour des sources de données distribuées ; et (iii) pour des données en flux. Toutes ces

approches nécessitent de parcourir les données. Par conséquent, elles sont peu adaptées aux sources de données distantes du Web sémantique, pour lesquelles l'accès se fait à distance via un point d'accès SPARQL. La difficulté de la découverte des patterns structurels sur des sources distantes réside essentiellement dans le fait qu'on ne peut pas parcourir les données, le seul accès se fait à travers des requêtes. De plus, le serveur peut imposer des restrictions d'accès comme un temps limité pour l'exécution d'une requête (timeout) ou la limitation du nombre de requêtes envoyées pour ne pas engorger le réseaux.

Dans ce chapitre, nous proposons une approche qui permet de découvrir le schéma versionné d'une source de données distante en découvrant les versions des types qu'elle contient. Cette approche doit faire face essentiellement à un problème combinatoire ainsi qu'aux restrictions imposées par les sources de données. L'accès à ces données est possible via des points d'accès distants pour des requêtes SPARQL [1].

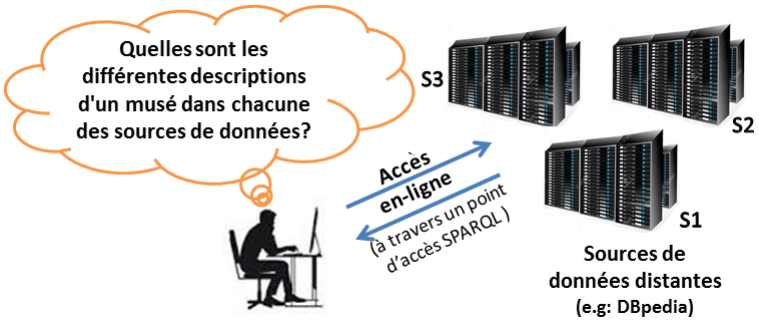


FIGURE 4.1 – Accès en ligne à des sources de données distantes.

La figure 4.1 montre un exemple où un utilisateur souhaite trouver les différentes descriptions d'un musée dans trois sources de données distantes sur le Web (S1, S2 et S3). Plusieurs descriptions peuvent être trouvées dans chaque source. Nous supposons que la recherche est effectuée en utilisant un simple ordinateur de bureau avec une puissance de calcul limitée et qu'il y a un temps limité pour répondre à la requête de l'utilisateur. Dans le contexte d'une source de données distante sur le Web, l'utilisateur ne peut pas parcourir les données ; son seul accès se fait via des requêtes envoyées au serveur Web qui gère la source de données.

Rappelons que nous représentons graphiquement le schéma d'une source de données par un ensemble de types et de liens entre eux. Chaque type représente un groupe d'instances avec la même déclaration de type dans la source de données. Un lien représente une propriété, soit entre un type et un littéral, soit entre deux types. Sur le schéma, un lien p d'un type t_i qui n'a pas de type cible indique qu'une instance du type t_i peut avoir la propriété p pour laquelle la valeur est un littéral. Un lien p d'un type t_i à un autre type t_j indique qu'une instance du type

1. SPARQL Query Language : http ://www.w3.org/TR/rdf-sparql-query/

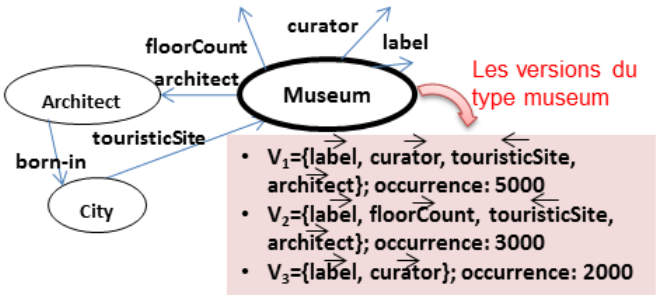


FIGURE 4.2 – Exemple d'un schéma versionné pour une source de données RDF

t_i peut avoir la propriété p pour laquelle la valeur est une instance du type t_i. Ce lien représente une propriété pour laquelle le domaine est t_i et le co-domaine est t_j. Une telle propriété est déclarée dans la source de données par les deux triplets (p $rdfs$:$domain$ t_i) et (p $rdfs$:$range$ t_j).

Nous proposons de décrire le contenu d'une source de données par un schéma versionné. la figure 4.2 montre un exemple partiel de schéma versionné pour la source S_1. Le type $Museum$ possède trois versions dans cette source de données : $v_1 = \{\overrightarrow{label}, \overrightarrow{curator}, \overleftarrow{touristicSite}, \overrightarrow{architecte}\}$; $v_2 = \{\overrightarrow{label}, \overleftarrow{floorCount}, \overleftarrow{touristicSite}, \overrightarrow{architecte}\}$ et $v_3 = \{\overrightarrow{label}, \overrightarrow{curator}\}$. Les versions de types montrent quelles propriétés surviennent ensemble dans la source de données, et idéalement le nombre d'instances pour lesquelles cette co-occurrence se produit.

La description d'un tel schéma versionné pourrait être utile pour un utilisateur dans diverses tâches de gestion de données. Une description des différentes structures d'un type et le nombre d'occurrences pour chacune d'elles aide l'utilisateur à formuler la requête la plus appropriée afin d'obtenir l'information dont il a besoin.

Lorsqu'une requête est émise sur plusieurs sources de données, la décomposition des requêtes est un problème clé, ainsi que la recherche du plan d'exécution optimal comme présenté dans [21]. L'ensemble des versions de types pour chaque source pourrait aider à décomposer la requête et à identifier les sous-requêtes à envoyer aux sources concernées. Le nombre d'occurrences des versions pourrait être utile pour optimiser les plans d'exécution en ordonnant les sous-requêtes selon la sélectivité de leurs propriétés (moins une propriété est probable, plus elle est sélective). En outre, le nombre d'occurrences de chaque version permet d'estimer le coût d'un plan d'exécution.

Le reste de ce chapitre est organisé comme suit. Nous présentons le principe de base de la découverte des versions et les défis à relever dans la section 4.2. Dans la section 4.3, nous présentons *SchemaDecrypt*, notre approche pour découvrir les versions d'un type en ligne. Nous proposons d'améliorer les performances de cette approche par une exploration parallèle des versions potentielles avec *SchemaDecrypt++* dans la section 4.4. Nous discutons le coût de notre approche

dans la section 4.5. Dans la section 4.6, nous présentons la méthodologie utilisée pour évaluer l'approche et les résultats obtenus sur *DBpedia*. Une conclusion est fournie dans la section 6.10.

4.2 Principes de base et défis

Pour trouver le schéma versionné d'une source de données, nous devons trouver les différentes versions de chaque type dans cette source de données. Dans cette section, nous présentons d'abord un ensemble de définitions dans la section 4.2.1 Nous formulons ensuite le problème de la découverte de version comme un problème combinatoire en section 4.2.2. Enfin, dans la section 4.2.3 nous présentons les restrictions qui peuvent être imposées par les sources de données distantes.

4.2.1 Définitions

Les différentes structures des instances d'un type représentent les versions possibles de ce type. Nous définissons une version d'un type et l'ensemble de ses versions comme suit.

Définition (Version). Une version v_i d'un type t est un ensemble de propriétés qui décrivent certaines instances de t ayant exactement les mêmes propriétés. Pour que $v_i = \{p_1, ..., p_n\}$ soit une version d'un type t, celle-ci doit contenir au moins une instance e telle que :
- $\forall p_j \in v_i$: p_j est une propriété décrivant e ;
- et, $\forall p_k$ de t décrivant e : $p_k \in v_i$.

Définition (Ensemble de versions d'un type). L'ensemble des versions V d'un type t représente l'ensemble des structures possibles de t. Il s'agit de l'ensemble de toutes les versions possibles du type t, tel que :
- ($\forall e$ du type t, $\exists v_i \in V$) et ($\forall v_i \in V$, $\exists e$ du type t) : e est décrit par toutes les propriétés de v_i et uniquement par les propriétés de v_i.

Le problème de trouver les versions d'un type est lié au problème de la co-occurrence des propriétés [37]. En effet, une version d'un type représente la co-occurrence de son ensemble de propriétés pour au moins une instance dans le type. Cependant, trouver les versions d'un type est plus général et plus complet que de trouver la co-occurrence des propriétés, car trouver les versions consiste à trouver toutes les co-occurrences de propriétés et pas seulement les plus fréquentes.

Afin de découvrir les versions d'un type pour une source de données distante et massive, nous proposons de générer un ensemble de requêtes utilisant les propriétés de ce type. Dans la source de données, une propriété p est déclarée pour le type t par ces triplets spécifiques : (p *rdfs:domain* t) ou (p *rdfs:range* t).

En plus des propriétés déclarées pour un type t, une instance de ce type peut être décrite par les propriétés déclarées pour les super-types ou les sous-types de t. En effet, les types d'une source de données RDF(S) sont organisés en hiérarchie, comme montré dans l'exemple de la figure 4.3, extrait de *DBpedia*, qui représente la hiérarchie contenant le type *Museum*. Par conséquent, notre approche doit tenir compte des propriétés des super-types et des sous-types lorsqu'elle recherche les différentes versions d'un type. Nous définissons l'ensemble des types associés à un type comme suit.

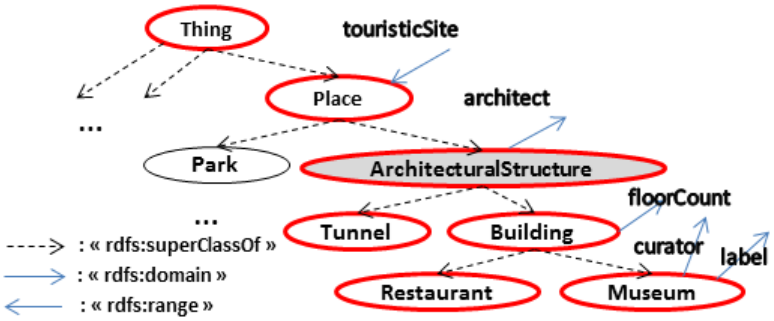


FIGURE 4.3 – La hiérarchie des types du type *Museum* dans *DBpedia*.

Définition (Ensemble de types associés). L'ensemble des types T associés à un type t est l'ensemble constitué de t, tous ses super-types et tous ses sous-types.

Par exemple, l'ensemble des types associés au type *ArchitecturalStructure* dans la figure 4.3 est : $T = \{$ *ArchitecturalStructure, Tunnel, Building, Museum, Restaurant, Place, Thing*$\}$.

Soit P l'ensemble des propriétés déclarées dans la source de données pour les types associés à un type donné t. Pour trouver les différentes versions de t, nous générons d'abord une version candidate à partir des propriétés dans P, puis nous interrogeons la source de données pour obtenir le nombre d'instances ayant les mêmes propriétés que cette version candidate. Nous définissons l'ensemble des propriétés associées à un type comme suit.

Définition (Ensemble de propriétés associées). L'ensemble des propriétés P associées à un type t est l'ensemble de propriétés déclarées pour tous ses types associés dans l'ensemble T, comme suit :

- $\forall\ t' \in T$: si $\exists\ (p_i\ rdfs{:}domain\ t')$ ou $\exists\ (p_i\ rdfs{:}range\ t')$ dans la source de données, alors $p_i \in P$.

Par exemple, l'ensemble des propriétés associées au type *Museum* dans la figure 4.3 est : $P = \{\overrightarrow{label}, \overrightarrow{curator}, \overrightarrow{floorCount}, \overrightarrow{architect}, \overleftarrow{touristicSite}\}$. Étant

donné cet ensemble de propriétés associées, nous définissons maintenant une version candidate comme suit.

Définition (Version candidate). Une version candidate v_i d'un type t est une combinaison de k propriétés de l'ensemble P des propriétés associées au type t, avec $k \leq |P|$; $v_i = \{p_1, ..., p_k\}$ est une version candidate pour le type t si :
- $\forall p_j \in v_i : p_j \in P$.

Par exemple, pour tester la version candidate $v_1 = \{\overrightarrow{label}, \overrightarrow{floorCount}, \overleftarrow{touristicSite}\}$ pour le type *Museum*, nous interrogeons la source de données distante en utilisant la requête SPARQL suivante :
- "*Select (COUNT(DISTINCT(?e)) as ?Nb) WHERE*
  - *{ ?e rdf:type Museum . ?e label ?n . ?e floorCount ?m . ?y touristicSite ?e}*" ; (**Requête 1**).

Si le nombre d'instances est positif, la version candidate est validée et considérée comme une version du type.

4.2.2 La découverte de versions, un problème combinatoire

Lorsque le nombre de propriétés d'un type est grand, cette approche présente un problème combinatoire. De plus, des contraintes d'accès à la source de données peuvent être imposées par le serveur, comme un temps limité pour répondre à une requête ou une perte de priorité de traitement si un grand nombre de requêtes est envoyé à la source.

Pour trouver les versions d'un type, nous proposons de générer les combinaisons possibles à partir de l'ensemble de propriétés associé à un type afin de former des versions candidates.

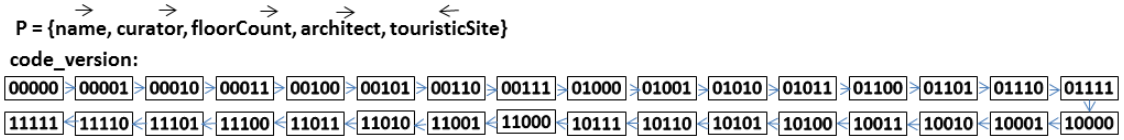


FIGURE 4.4 – Recherche exhaustive des versions du type *Museum*.

Soit $P = \{p_1, p_2, ...p_n\}$ l'ensemble de propriétés associées à un type t. Les versions candidates sont toutes les combinaisons d'éléments k de P, avec k variant de 1 à n, ce qui représente 2^n combinaisons. Les versions validées sont celles pour lesquelles il existe des instances du type conformes à cette version, et le nombre d'instances représente le nombre d'occurrences de la version. Dans notre approche, un code est associé à chaque version, où chaque propriété de P

correspond à un bit ; le *code_version* est défini comme suit.

Définition (*code_version*)**.** Un *code_version* est une codification binaire pour une version d'un type. Nous représentons par un bit dans le *code_version* chaque propriété p_i de l'ensemble P des propriétés associées à un type, comme suit :

- $bit(p_i) = 1$ si p_i est présent dans la version ;
- $bit(p_i) = 0$ sinon.

Le nombre d'occurrences de chaque version est calculé en fonction du nombre d'instances du type conformes à cette version en utilisant une requête *Count* comme dans la **Requête 1**.

La figure 4.4 représente le principe de base pour découvrir les versions du type *Museum*, qui consiste à tester toutes les versions possibles. Cela peut être comparé au processus de recherche d'une clé en cryptanalyse [82] : l'approche de base est une recherche exhaustive des versions, celle-ci correspond à une attaque par force brute. Le nombre de versions candidates générées à partir de l'ensemble des propriétés P du type *Museum*, qui a 5 propriétés, est $2^5 = 32$. Plus généralement, la recherche exhaustive de versions pour un type avec n propriétés nécessite de générer et tester 2^n versions candidates. Cela peut vite représenter un nombre astronomique lorsque le nombre de propriétés devient important. À titre d'exemple, le nombre de propriétés du type *Museum* dans *DBpedia* est de 291 et le nombre moyen de propriétés d'un type dans la source de données *DBpedia* est de 150 [83]. Pour donner un ordre de grandeur de ce nombre : il existe 2^{150} versions candidates à tester, le serveur en ligne de *DBpedia*[2] peut tester au maximum 15 requêtes par seconde [83], le meilleur temps estimé pour le traitement de 2^{150} requêtes est donc de $2^{150}/15$ secondes $\approx 2^{146}$ secondes, sachant que 1 an = 31 536 000 secondes $\approx 2^{25}$ secondes ; 2^{146} secondes $\approx 2^{121}$ années $\approx 10^{36}$ années. Ceci est évidemment impossible à tester.

En plus de ce problème combinatoire, deux difficultés supplémentaires apparaissent dans notre contexte :

- Toutes les versions d'un type doivent être découvertes, mais nous ne connaissons pas *a priori* leur nombre et donc quand arrêter la recherche ;
- Des recouvrements entre les versions peuvent se produire, par exemple, on peut voir que l'ensemble des instances d'une version candidate $v_i = \{\overrightarrow{label}, \overrightarrow{curator}, \overleftarrow{touristicSite}\}$ est inclus dans l'ensemble des instances de la version candidate $v_j = \{\overrightarrow{label}, \overrightarrow{curator}\}$, par conséquent, lorsque la source de données est interrogée pour obtenir le nombre d'instances ayant les propriétés de la version v_i, la réponse comprendra les instances ayant les propriétés des deux versions v_i et v_j.

2. Accès en ligne de *DBpedia* : http ://wiki.dbpedia.org/OnlineAccess

4.2.3 Restrictions de la source de données

Notre objectif est de trouver les versions d'un type d'une source de données distante. Cela signifie que nous ne pouvons pas parcourir les données, mais que nous pouvons seulement interroger le serveur qui gère la source de données en ligne. Toutefois, le serveur Web a généralement des restrictions sur l'accès aux données. En effet, certaines requêtes génèrent des exceptions lorsqu'une restriction du serveur Web n'est pas respectée. Ces restrictions sont en place pour assurer que tout le monde a la même chance d'interroger les données du serveur et également pour se protéger des requêtes mal écrites et des robots. Les restrictions sont les suivantes :

- Une taille maximale limitée pour les résultats : un serveur Web limite la taille maximale des données renvoyées pour éviter d'engorger le réseau ;
- Un temps limité pour le traitement d'une requête : lorsque le nombre de propriétés contenues dans une requête est important, le serveur déclenche une exception de type timeout, en considérant que le traitement de la requête risque de prendre beaucoup de temps ;
- Un nombre limité de requêtes : un serveur Web peut avoir des listes de contrôle d'accès HTTP qui permettent à l'administrateur d'indiquer une limite pour certaines adresses IP. Si un nombre important de requêtes est envoyé au serveur, cela peut entraîner une perte de priorité de traitement et même un refus d'accès au serveur.

Par exemple, pour la source de données *DBpedia*, son serveur en ligne est configuré pour traiter des requêtes avec une fenêtre de délai d'attente permettant un temps d'exécution maximum de 120 secondes et une taille de résultat maximale de 2000 lignes [83]. En outre, si un nombre important de requêtes sont envoyées au serveur, cela peut entraîner la perte de priorité de traitement.

Comme notre approche interroge le serveur pour connaître le nombre d'instances, elle n'est pas affectée par la restriction sur la taille maximale du résultat. Toutefois, comme le nombre de propriétés dans une requête peut être important pour former une version candidate, l'estimation du coût de la requête peut dépasser 120 secondes. De plus, pour trouver toutes les versions d'un type, nous testons plusieurs versions candidates via des requêtes, ce qui peut entraîner une perte de priorité de traitement.

Notre approche de découverte des versions d'un type doit donc respecter les exigences suivantes :

- Utiliser un nombre minimum de propriétés dans une requête pour réduire son temps d'exécution ;
- Envoyer un nombre minimum de requêtes au serveur pour ne pas perdre la priorité de traitement.

4.3 Découverte en ligne des versions d'un type

Nous proposons l'approche *SchemaDecrypt* pou trouver le schéma versionné d'une source de données distante, qui consiste à trouver les différentes versions de ses types. Celle-ci est fondée sur la construction d'un profil de type probabiliste qui permet : (i) de guider l'exploration des versions candidates en testant d'abord les versions les plus probables ; (ii) de réduire l'espace de recherche des versions candidates et (iii) de définir un critère d'arrêt pour l'exploration. Nous proposons également de réduire l'ensemble de propriétés associées à combiner en fonction du profil du type. Pour réduire le nombre de versions candidates et le nombre de requêtes envoyées à la source de données, nous proposons de déduire des règles d'occurrences entre les propriétés du type. Ces règles sont exploitées lors de la génération dynamique des versions candidates.

Dans cette section, nous présentons le processus de construction du profil probabiliste de type dans la section 4.3.1, puis la réduction de l'ensemble des propriétés associées à un type dans la section 4.3.2. Nous présentons notre approche pour déduire des règles entre les propriétés du type dans la section 4.3.3. La génération dynamique des versions candidates est présentée dans la section 4.3.4, puis l'exploitation des règles lors de la génération dynamique des versions candidates est présentée dans la section 4.3.5. Une étude de cas utilisant *SchemaDecrypt* est présentée dans la section 4.3.6.

4.3.1 Construction d'un profil de type probabiliste

Comme les instances du même type ne suivent pas nécessairement la description exacte du type, nous définissons un profil probabiliste pour un type comme suit.

Définition (Profil probabiliste d'un type). Un profil probabiliste TP d'un type t est formé par l'ensemble P des propriétés associées au type t avec leurs probabilités :

- $TP = \{(p_1, \alpha_1), ..., (p_n, \alpha_n)\}$, $p_i \in P$, où α_i représente la probabilité qu'une instance de t ait la propriété p_i.

Comme dans le chapitre 3, la probabilité α_i associée à une propriété p_i dans le profil du type t représente le nombre d'instances du type t pour lesquelles p_i est défini sur le nombre total d'instances dans t. La différence entre ce profil et celui présenté dans la chapitre 3, réside dans sa construction. Dans ce chapitre, le profil d'un type est construit en interrogeant la source sur chaque propriété de l'ensemble des propriétés associées au type.

Notez qu'une propriété peut être entrante, comme la propriété $\overleftarrow{touristicSite}$, ou sortante, comme la propriété $\overrightarrow{curator}$ de la figure 4.2. Le domaine et le co-

domaine d'une propriété sont donc importants lors de l'interrogation de la source de données. Pour créer le profil d'un type t, la source de données est interrogée sur chaque propriété entrante p_i de l'ensemble des propriétés P associées à un type t comme dans la requête 2 (a) et sur chaque propriété sortante p_j comme dans la requête 2 (b)

- "*Select (COUNT(DISTINCT(?e)) as ?propertyOccur) WHERE*
  - *{ ?e rdf:type t . ?n p_i ?e }*" ; (**Requête 2 (a)**)
- "*Select (COUNT(DISTINCT(?e)) as ?propertyOccur) WHERE*
  - *{ ?e rdf:type t . ?e p_j ?n }*" ; (**Requête 2 (b)**)

PropertyOccur dans les requêtes 2 (a) et 2 (b) présente le nombre d'occurrences d'une propriété pour décrire les instances d'un type t. La probabilité d'une propriété représente la valeur de *PropertyOccur* divisée par le nombre d'instances du type t.

Il faut noter que ces requêtes ne nécessitent pas de parcourir les données, mais uniquement d'accéder à l'index défini sur ces données.

4.3.2 Réduction du nombre de propriétés

Nous proposons de réduire le nombre de combinaisons pour former des versions candidates en diminuant le nombre de propriétés à tester. Étant donné P, l'ensemble des propriétés associées à un type, certaines propriétés de P se produisent toujours ensemble pour décrire les instances du type, et nous désignons ces propriétés par *propriétés co-occurrentes*.

Nous proposons d'identifier les propriétés co-occurrentes et de les représenter comme une propriété unique, ce qui réduira le nombre de propriétés à tester et donc le nombre de combinaisons et le nombre de versions candidates à explorer.

L'**algorithme 8** représente notre approche pour détecter les propriétés co-occurrentes. Deux propriétés p_i et p_j peuvent avoir les mêmes occurrences si elles ont la même probabilité. Par conséquent, nous ne testons que les paires de propriétés du profil de type qui ont la même probabilité. Deux propriétés ont effectivement les mêmes occurrences si, en plus, la probabilité d'une instance dans le type décrite par les deux propriétés est égale à la probabilité d'une des deux propriétés. Pour calculer $\alpha_{i,j}$, la probabilité des propriétés p_i et p_j de décrire en même temps une instance d'un type t, nous interrogeons la source de données pour obtenir le nombre d'instances dans le type décrite par p_i et p_j, comme suit :

- "*Select (COUNT(DISTINCT(?e)) as ?x) WHERE*
  - *{ ?e rdf:type t . ?e p_i ?n. ?e p_j ?b }*" ; (**Requête 3**)

Ensuite, x est divisé par le nombre d'instances du type t pour calculer $\alpha_{i,j}$, la probabilité de p_i et p_j. L'orientation d'une propriété (entrante ou sortante) est prise en compte lors de la formulation de la requête.

Comme décrit dans l'**algorithme 8**, nous représentons chaque sous-ensemble

Algorithm 8: Propriétés co-occurrentes

Input : L'ensemble des propriétés associées P, le profil du type TP
Output: Tous les sous-ensembles de propriétés avec les même occurrences E_i

```
1  for ∀p_i ∈ P do
2      for ∀p_j ∈ P avec p_i != p_j do
3          if (α_i = α_j dans TP) then
4              α_{i,j} = La probabilité d'occurrences de p_i et p_j;
5              if (α_{i,j} = α_i) then
6                  if (∃ p_i ∈ sous-ensemble de propriétés E_i) then
7                      ajouter p_j à E_i;
8                  else
9                      Créer un sous-ensemble de propriétés E_i avec p_i et p_j;
10                 end
11             end
12         end
13     end
14 end
```

E_i de propriétés ayant les mêmes occurrences par une seule propriété dans notre approche. En outre, les propriétés dans le profil de type avec une probabilité égale à 1 sont dans toutes les versions et, par conséquent, il n'est pas utile de les considérer pour tester les versions candidates. De la même manière, les propriétés du profil de type ayant une probabilité égale à 0 ne doivent pas être considérées car elles ne sont présentes dans aucune version du type. Nous définissons l'ensemble réduit des propriétés associées à un type comme suit.

Définition (Ensemble réduit de propriétés associées). Un ensemble réduit E de l'ensemble P des propriétés associées à un type est formé par des propriétés de P telles que $\forall p_i \in P$, $p_i \in E$ si :

- $\nexists\, p_j \in E$, et p_j et p_i sont co-occurrentes ;
- et $\alpha_i \in]0, 1[$.

4.3.3 Découverte des règles d'occurrences

Pour réduire le nombre de requêtes émises et minimiser le nombre de propriétés dans une requête, nous proposons de détecter des règles d'occurrences entre les propriétés associées à un type.

Nous introduisons deux types de règles entre les propriétés d'un type : (i) des règles d'inclusion ($p_i \Rightarrow p_j$) qui indiquent que l'ensemble des instances caractérisées par la propriété p_i est inclus dans l'ensemble des instances caractérisées

par la propriété p_j et (ii) des règles d'exclusion ($p_i \mid p_j$) qui indiquent que les propriétés p_i et p_j ne décrivent jamais les mêmes instances, donc les occurrences de p_i sont disjointes des occurrences de p_j.

Ces règles permettent, entre autres, de savoir localement si certaines versions candidates peuvent être des versions réelles du type, sans envoyer les requêtes associées à la source de données distante. En effet, si chaque fois qu'une propriété p_i se produit alors la propriété p_j se produit, on peut en déduire que $p_i \Rightarrow p_j$. Cela signifie que les versions candidates avec p_i et sans p_j n'existent pas et il n'est donc pas utile d'envoyer les requêtes associées à la source de données. Un autre type de règle est défini lorsque les propriétés p_i et p_j ne se produisent jamais ensemble, ce qui est représenté par un *Not AND (NAND)*. Ce lien est noté $p_i \mid p_j$ pour toutes les versions du type. Cela signifie que les versions candidates avec p_i et p_j n'existent pas et, par conséquent, qu'il n'est pas nécessaire d'envoyer les requêtes associées à la source de données.

Soient E l'ensemble réduit de propriétés associées à un type t, α_i (resp. α_j) la probabilité d'une propriété p_i (resp. p_j) de décrire une instance d'un type t dans le profil de type et $\alpha_{i,j}$ la probabilité des propriétés p_i et p_j pour décrire une instance de t. Nous proposons de rechercher des règles d'inclusion et d'exclusion entre les propriétés du type t comme suit.

Règles d'inclusion. Pour déterminer les règles d'inclusion entre les propriétés d'un type, nous ne testons pas chaque paire de propriétés associées au type, mais seulement celles pour lesquelles une inclusion est possible. En effet, une inclusion est possible entre deux propriétés si la probabilité d'une d'entre elles est supérieure à la probabilité de l'autre. Nous déterminons les règles d'inclusion dans un type comme suit :

- $\forall\ p_i, p_j \in E$, si $\alpha_i\ != \alpha_j$ dans le profil de type, alors :
  - Calculer $\alpha_{i,j}$ comme dans la requête (3)
  - Si ($\alpha_{i,j} = \alpha_i$) alors ($p_i \Rightarrow p_j$) est une règle d'inclusion
  - Sinon, si ($\alpha_{i,j} = \alpha_j$) alors ($p_j \Rightarrow p_i$) est une règle d'inclusion

Notez qu'une règle d'inclusion n'est pas une dépendance fonctionnelle. En effet, une dépendance fonctionnelle exprime une contrainte sur les valeurs des propriétés alors que dans notre approche, une règle d'inclusion exprime une contrainte sur la présence des propriétés.

Règles d'exclusion. Pour déterminer les règles d'exclusion entre les propriétés d'un type, nous ne testons pas chaque paire de propriétés associées à un type, mais seulement celles pour lesquelles une exclusion est possible. En effet, une exclusion est possible entre deux propriétés si l'addition des probabilités des deux propriétés est inférieure ou égale à 1. Nous déterminons les règles d'exclusion dans un type comme suit :

- $\forall\ p_i, p_j \in E$, si $\alpha_i + \alpha_j \leq 1$ dans le profil de type, alors :
  - Calculer $\alpha_{i,j}$ comme dans la requête (3)
  - Si $(\alpha_{i,j} = 0)$ alors $(p_i \mid p_j)$ est une règle d'exclusion

4.3.4 Génération dynamique des versions candidates

Afin de trouver les différentes versions d'un type, nous générons progressivement des versions candidates à partir de l'ensemble réduit de propriétés E, jusqu'à trouver toutes les versions du type, comme décrit dans l'**algorithme 9**. Nous utilisons notre *code_version* comme une codification binaire d'une version candidate, où chaque propriété dans E est représentée par un bit. *Code_version* est initialisé à sa valeur maximale $(2^{|E|} - 1)$ et il est décrémenté jusqu'à ce que toutes les versions du type soient découvertes. Cela permet de tester les versions candidates de manière ordonnée et d'éviter de construire le graphe de toutes les combinaisons *a priori*, ce qui optimise la mémoire utilisée pendant le processus.

L'initialisation du *code_version* à sa valeur maximale permet de commencer par tester et découvrir les versions qui contiennent le plus grand nombre de propriétés afin d'obtenir le nombre exact de leurs occurrences dans la source de données. En effet, certaines inclusions entre les versions peuvent se produire, et tester celles avec le plus grand nombre de propriétés d'abord évitera de compter plusieurs fois les mêmes instances. Par exemple, nous pouvons voir que les instances de la version $v_j = \{\overrightarrow{label}, \overrightarrow{curator}, \overrightarrow{touristicSite}\}$ sont incluses dans l'ensemble des instances de la version $v_i = \{\overrightarrow{label}, \overrightarrow{curator}\}$, et lorsque la source de données est interrogée pour obtenir le nombre d'instances ayant les propriétés de la version v_i, la réponse inclura les occurrences ayant les propriétés des deux versions v_i et v_j. Nous considérons le nombre d'occurrences d'une version $Occurrences(v_i)$ comme le nombre d'instances n'ayant que les propriétés exactes de la version et $Count(v_i)$ le nombre d'instances renvoyé par la source de données. Soit V l'ensemble des versions validées quand v_i est testée. Le nombre d'occurrences de v_i est calculé comme dans la formule 4.1.

$$Occurrences(v_i) = Count(v_i) - \sum_{\forall v_j \in V \wedge v_i \subset v_j} Occurrences(v_j) \tag{4.1}$$

La version candidate v_i est validée et ajoutée à l'ensemble des versions validées V si $Occurrences(v_i) > 0$.

Chaque fois qu'une version candidate est validée, nous mettons à jour le profil du type afin que les probabilités reflètent uniquement les versions qui n'ont pas encore été découvertes. Cela permettra de définir le critère d'arrêt, qui sera atteint lorsque toutes les probabilités des propriétés dans le profil du type sont à 0, ce qui signifie que toutes les versions du type ont été découvertes (voir lignes 14 et 15 de l'**algorithme 9**). Le profil du type est mis à jour à l'aide de la

Algorithm 9: Génération dynamique des versions candidates

Input : Le profil de type TP, l'ensemble réduit de propriétés E, l'ensemble des règles R

Output: L'ensemble des versions validées V avec le nombre d'occurrences de chaque version

```
1  code_version = 2^|E| − 1;
2  Ordonner l'ensemble E de la propriété la plus probable à la moins
     probable;
3  while (E ≠ ∅) do
4      Construire la version candidate v_i à partir de E formée par les
         propriétés avec bit(p_j) = 1 dans code_version;
5      if (∀r ∈ R : v_i respecte r) then
6          Soit q la requête correspondante à v_i;
7          Réduire la taille de q selon les règles d'inclusion dans R;
8          Ordonner les propriétés de q de la plus sélective à la moins sélective;
9          Envoyer la requête q à la source de données;
10         if (Count(v_i) > 0) then
11             Occurrences(v_i) = Count(v_i) − Σ_{∀v_j∈V∧v_i⊂v_j} Occurrences(v_j);
12             if (Occurrences(v_i) > 0) then
13                 Ajouter v_i à V et sauvegarder Occurrences(v_i);
14                 MiseAJourProfilType(TP, v_i);
15                 Supprimer de E la propriété avec une probabilité égale à 0
                     dans TP;
16                 if (|E| a changé) then
17                     code_version = 2^|E|;
18                     Ordonner l'ensemble E de la propriété la plus probable à
                         la moins probable;
19                 end
20             end
21         end
22         code_version = code_version - 1;
23     else
24         code_version = code_version - Saut(code_version, E, R);
25     end
26 end
```

Algorithm 10: Mise à jour profil de type

Input : Le profil de type TP, une version validée v
Output: TP à jour

```
1 for ∀p ∈ v do
2 |   Soit α la probabilité de p dans TP;
3 |   α ← α − Occurrences(v) / nbInstancesClass;
4 end
```

fonction **MiseAJourProfilType** présentée dans l'**algorithme 10**, qui met à jour la probabilité de chaque propriété de la version découverte.

Si la probabilité d'une propriété passe à 0 dans le profil du type, cela signifie que toutes les versions qui contiennent cette propriété ont été trouvées. Dans ce cas, la propriété est supprimée de E, l'ensemble de propriétés à partir duquel les versions sont générées (voir la ligne 15 de l'**algorithme 9**). Nous réinitialisons le *code_version* pour avoir un nombre de bits égal au nombre de propriétés restantes dans E.

Afin de tester d'abord la version la plus probable, nous ordonnons l'ensemble E de la propriété la plus probable à la moins probable (voir lignes 2 et 18 de l'**algorithme 9**). Dans la requête, nous ordonnons les propriétés de la moins probable à la plus probable afin d'avoir un temps de réponse optimal des sources de données en partant de la propriété la plus sélective (ayant la probabilité la plus faible) comme décrit dans la ligne 8 de l'**algorithme 9**.

Pour chaque version candidate générée, *SchemaDecrypt* vérifie si cette version est conforme aux règles. Une version candidate peut avoir des instances dans la source de données si elle respecte toutes les règles. Si une version candidate est conforme à toutes les règles, nous générons la requête correspondante et utilisons les règles d'inclusion pour réduire sa taille. Toutefois, si une version candidate viole une règle, l'algorithme passe à la prochaine version candidate qui ne viole aucune règle. Nous exploitons les règles d'occurrences à des fins différentes dans l'**algorithme 9** : (i) pour tester si une version candidate est possible (voir la ligne 5) ; (ii) pour réduire la taille d'une requête (voir la ligne 7) et (iii) pour faire des sauts vers la prochaine version candidate qui ne viole pas les règles (voir la ligne 24). Dans la prochaine section, nous présentons chacune de ces étapes.

4.3.5 Exploitation des règles d'occurrences

Nous proposons d'exploiter les règles d'inclusion et d'exclusion lors de la découverte des versions d'un type dans l'**algorithme 9**. Soit $r = p_i \ \varphi \ p_j$ une règle, où $\varphi \in \{\Rightarrow, \mid\}$. Si r est une règle d'inclusion, cela signifie que si une instance du type est décrite par la propriété p_i, elle est également décrite par la propriété p_j. Par conséquent, la propriété p_j n'est pas nécessaire dans une requête qui com-

prend la propriété p_i, et la propriété p_j peut donc être supprimée de la requête pour avoir un meilleur temps de réponse. En effet, plus il y a de propriétés dans une requête, plus le temps de réponse est élevé, ce qui peut provoquer un *timeout*.

Si une version candidate n'est pas conforme à une règle, il n'est pas nécessaire d'envoyer sa requête à la source de données, car celle-ci répondra par un nombre d'instance égale à 0 : en effet, si r est une règle d'inclusion, il n'y a pas d'instances dans la source de données qui correspondent à un *code_version* avec un bit(p_i) = 1 et bit(p_j) = 0 ; si r est une règle d'exclusion, il n'y a pas d'instances dans la source de données qui correspondent à un *code_version* avec un bit(p_i) = 1 et bit (p_j) = 1.

Si une version candidate n'est pas conforme à une règle, certaines des versions suivantes pourraient également ne pas être conformes à la règle. Dans ce cas, il est plus efficace de passer directement à la prochaine version qui respecte la règle, mais le problème est de savoir comment trouver cette version ?

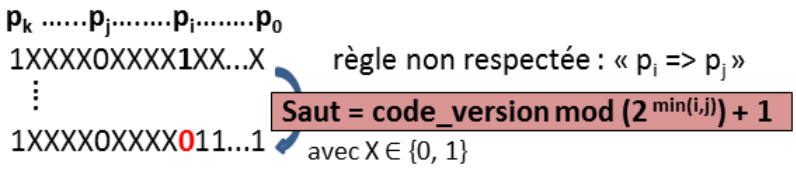


FIGURE 4.5 – Un saut dans un *code_version*.

La prochaine version conforme à la règle est la première qui modifie soit $bit(p_i)$ soit $bit(p_j)$. Le premier bit qui sera modifié en décrémentant le *code_version* est le minimum entre i et j, comme le montre la figure 4.5. La prochaine version conforme à la règle correspond au *code_version* calculé avec un *Saut* décrit dans la formule 4.2.

$$code\_version = code\_version - Saut \tag{4.2}$$

Avec une valeur du *Saut* calculé comme décrit dans la formule 4.3.

$$Saut = code\_version(2^{min(i,j)}) + 1 \tag{4.3}$$

4.3.5.1 Traitement d'une règle

Soit $r = p_i \ \varphi \ p_j$ une règle, avec $\varphi \in \{\Rightarrow , | \}$. Comme décrit précédemment, nous proposons d'exploiter cette règle comme suit :

- La propriété p_j est supprimée de la requête si $((\varphi = \Rightarrow) \wedge (\text{bit}(p_i) = 1) \wedge (\text{bit}(p_j) = 1))$
- Une version candidate est ignorée si elle n'est pas conforme à la règle, dans les cas suivants :
  - $(\varphi = \Rightarrow) \wedge (\text{bit}(p_i) = 1) \wedge (\text{bit}(p_j) = 0)$

- $(\varphi = | \wedge (\text{bit}(p_i) = 1) \wedge (\text{bit}(p_j) = 1)$
- Si une règle est violée, le prochain *code_version* à tester est calculé comme décrit dans la formule 4.2.

Malgré l'élimination d'une propriété dans une requête, sa probabilité dans le profil de type est mise à jour pour chaque version validée à laquelle elle appartient.

4.3.5.2 Sélection d'une règle

Il est possible qu'une version candidate ne respecte pas plusieurs règles, dans ce cas, quelle règle doit être considérée en premier ? Le problème ici est de savoir s'il existe un ordre optimal pour traiter les règles.

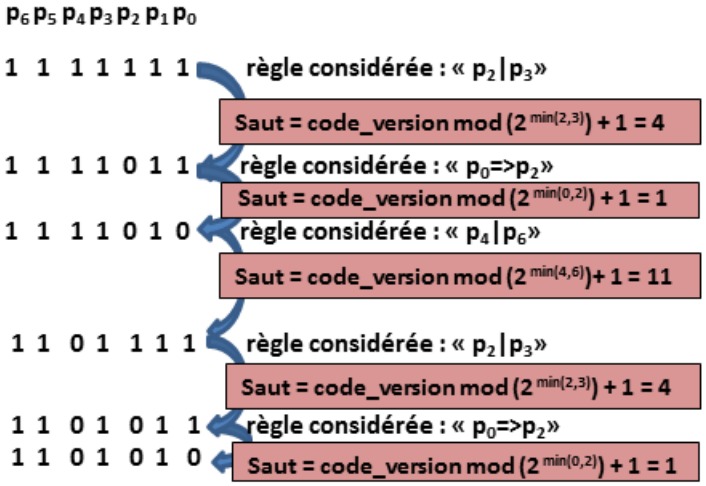


FIGURE 4.6 – Un exemple d'exploitation de plusieurs règles.

Considérons l'ensemble des règles $R = \{\ p_0 \Rightarrow p_2,\ p_2 \mid p_3,\ p_4 \mid p_6\ \}$. La figure 4.6 montre un exemple de versions qui violent plusieurs règles. Nous pouvons constater que la règle $p_4 \mid p_6$ annule les traitements des deux règles $p_0 \Rightarrow p_2$ et $p_2 \mid p_3$. Plus généralement, une règle $r = p_i\ \varphi\ p_j$, avec $\varphi \in \{\Rightarrow, \text{v}\}$, peut affecter le traitement de toute règle $r' = p_{i'}\ \varphi\ p_{j'}$ pour laquelle $min(i', j') < min(i, j)$. Ceci est dû au fait que toutes les propriétés à droite de l'index $min(i, j)$ sont remises à 1 après traitement de la règle r.

Afin d'éviter les traitements inutiles, les règles qui sont violées par la version correspondante au *code_version* courant doivent être ordonnées afin de ne pas annuler l'effet du traitement de règles précédentes. Cela nécessite de maximiser les premiers sauts en traitant d'abord les règles ayant le $min(i, j)$ le plus élevé. Soit $R = \{r_1, ...r_n\}$ un ensemble de règles, l'indice de la première propriété à changer, désigné par $Index$, est calculé comme dans la formule 4.4.

$$Index = Max_{k=1}^{n} Min(i, j); r_k = p_i \varphi p_j \tag{4.4}$$

la figure 4.7 montre l'exemple précédent avec une exploitation optimale des mêmes règles. Nous pouvons constater que, pour trois violations de règles, il existe trois traitements au lieu de cinq dans la figure 4.6.

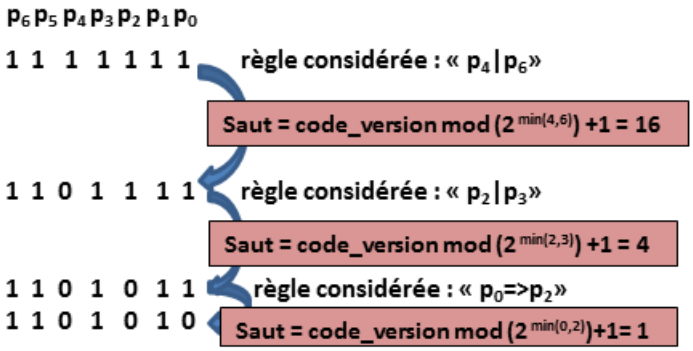


FIGURE 4.7 – L'exploitation optimale de plusieurs règles d'occurrences.

4.3.6 Étude de cas

Considérons l'exemple donné dans la section 6.1, et supposons qu'une source de données ait différentes instances de musées décrites comme suit : 5000 instances sont décrites par les propriétés $\{\overrightarrow{label}, \overrightarrow{Conservateur}, \overleftarrow{touristicSite}, \overrightarrow{architecte}\}$; 3000 propriétés sont décrites par les propriétés $\{\overrightarrow{label}, \overrightarrow{floorCount}, \overleftarrow{touristicSite}, \overrightarrow{architecte}\}$; et 2000 sont décrites par les propriétés $\{\overrightarrow{label}$ et $\overrightarrow{curator}\}$.

Pour trouver les différentes descriptions d'un musée à partir de cette source de données distante, *SchemaDecrypt* découvre d'abord l'ensemble de propriétés associées au type *Museum* : $P = \{\overrightarrow{label}, \overrightarrow{curator}, \overrightarrow{floorCount}, \overleftarrow{touristicSite}, \overrightarrow{architect}\}$. Ensuite, le profil du type est construit : $CP_{(Museum)} = \{(\overrightarrow{label}, 1), (\overrightarrow{floorCount}, 0.3), (\overrightarrow{curator}, 0.7), (\overleftarrow{touristicSite}, 0.8), (\overrightarrow{architect}, 0.8)\}$. Puis, *SchemaDecrypt* tente de réduire l'ensemble des propriétés associées en détectant celles ayant les mêmes occurrences : les propriétés $\overleftarrow{touristicSite}$ et $\overrightarrow{architect}$ ont les mêmes occurrences, donc elles vont être représentées par une propriété unique. La probabilité de la propriété $\overrightarrow{label}$ est à 1, par conséquent, cette propriété est ignorée au cours de la procédure. *SchemaDecrypt* construit l'ensemble réduit de propriétés associées : $E = \{\overrightarrow{curator}, \overrightarrow{floorCount}, \overleftarrow{touristicSite}\}$.

À partir de la source de données distante, *SchemaDecrypt* déduit les règles suivantes :

- $r_1 = \overrightarrow{floorCount} \Rightarrow \overleftarrow{touristicSite}$
- $r_2 = \overrightarrow{floorCount} \mid \overleftarrow{curator}$

L'ensemble E est ordonné selon les probabilités des propriétés dans le profil TP, ce qui donne :

- $E = \{\overleftarrow{touristicSite}, \overrightarrow{curator}, \overrightarrow{floorCount}\}$.

Le *code_version* correspondant est initialisé à sa valeur maximale : $code\_version = 2^{|E|} - 1 = (111)_2$. La version candidate qui correspond au *code_version* viole la règle r_2, donc la requête correspondante n'est pas envoyée et *SchemaDecrypt* passe à la prochaine version qui est conforme à la règle et qui est identifiée par *code_version* - *saut* = *code_version* - (*code_version* mod $(2^0) + 1) = (110)_2$. La version candidate est conforme à toutes les règles, donc la requête correspondante est générée et envoyée. La réponse de la source de données à cette requête est 5000. *SchemaDecrypt* ajoute à l'ensemble des versions validées V la version suivante :

- $v_1 = \{\overrightarrow{label}, \overrightarrow{curator}, \overleftarrow{touristicSite}, \overrightarrow{architect}\}$, avec un nombre d'occurrences de 5000.

Pour vérifier si toutes les versions sont trouvées et si le critère d'arrêt est atteint, *SchemaDecrypt* met à jour le profil du type pour prendre en compte les instances de la version v_1 ; le profil devient : $CP_{(Museum)} = \{(\overrightarrow{floorCount}, 0.3), (\overrightarrow{curator}, 0.2), (\overleftarrow{touristicSite}, 0.3)\}$. Comme aucune probabilité n'a atteint 0, le prochain *code_version* est 101_2. Selon la règle r_1, la source de données est interrogée avec la propriété $\overrightarrow{floorCount}$ seulement, car cette propriété est incluse dans la propriété $\overleftarrow{touristicSite}$. La réponse de la source de données à cette requête est de 3000. *SchemaDecrypt* ajoute à l'ensemble des versions validées V la version suivante :

- $v_2 = \{\overrightarrow{label}, \overrightarrow{floorCount}, \overleftarrow{touristicSite}, \overrightarrow{architect}\}$, avec un nombre d'occurrences de 3000.

SchemaDecrypt met à jour le profil du type pour tenir compte des exemples de la version v_2 comme suit : $CP_{(Museum)} = \{(\overrightarrow{floorCount}, 0), (\overrightarrow{curator}, 0.2), (\overleftarrow{touristicSite}, 0)\}$. Nous retirons de E les propriétés dont les probabilités ont atteint la valeur 0.

- $E = \{\overrightarrow{curator}\}$

Le prochain *code_version* est égal à $2^{|E|} - 1 = 1_2$. La version candidate correspondante est conforme à toutes les règles, donc la requête correspondante est construite et envoyée. La réponse de la source de données à cette requête est de 7000 pour la version $v_3 = \{\overrightarrow{label}, \overrightarrow{curator}\}$. Comme $v_3 \subset v_1$ alors Occurrences(v_3) = Count(v_3) - Occurrences(v_1) = 7000 - 5000 = 2000. *SchemaDecrypt* ajoute à l'ensemble des versions validées V la version suivante :

- $v_3 = \{\overrightarrow{label}, \overrightarrow{curator}\}$, avec un nombre d'occurrences de 2000.

SchemaDecrypt met à jour le profil de type, ce qui entraîne l'annulation de toutes les probabilités. Les propriétés dont les probabilités ont atteint la valeur 0 sont supprimées de E, par conséquent le critère d'arrêt est atteint avec $E = \emptyset$.

le type *Museum* a 3 versions. Pour trouver ces versions, *SchemaDecrypt* a généré 4 versions candidates et a envoyé 3 requêtes à la source de données qui ont été toutes validées, tandis qu'une recherche exhaustive aurait généré 32 versions candidates et envoyé 32 requêtes (voir la figure 4.4).

4.4 Découverte parallèle et en ligne des versions d'un type

L'approche que nous avons présentée dan la section 4.3 explore les versions d'un type de façon séquentielle. Dans cette section, nous introduisons *SchemaDecrypt* + +, qui permet une exploration parallèle de l'espace de recherche, et cela en exploitant les règles d'exclusion.

Les règles d'exclusion permettent de mettre en évidence les propriétés qui n'apparaissent jamais ensemble. *SchemaDecrypt* exploite ces règles pour réduire l'espace de recherche en ne testant pas les versions candidates qui ne les respectent pas. Dans *SchemaDecrypt*++, nous proposons en plus d'exploiter ces règles pour paralléliser les tâches de recherche des versions d'un type. Ceci est possible en formant des sous-ensembles de propriétés à partir de l'ensemble réduit des propriétés E (voir section 4.3.2) associées à un type, en utilisant les règles d'exclusion. Nous introduisons la notion de modèle pour représenter chaque sous-ensemble de versions qui caractérise un ensemble de versions candidates. Un modèle de versions est exclu des autres car chaque modèle de versions contient une propriété qui n'apparaît jamais avec certaines propriétés des autres modèles de versions à explorer en parallèle. Les modèles de versions peuvent être explorés en parallèle s'il n'y a pas de recouvrement possible entre leurs ensembles de versions candidates respectifs.

Le nombre de modèles de versions à explorer en parallèle à un instant donné ne doit pas être supérieur à la capacité de la source de données à traiter des requêtes en parallèle, sinon cela augmente le temps de réponse de la source de données. Dans ce cas, il est préférable de ne pas paralléliser certaines tâches d'exploration afin de ne pas surcharger la source de données et garder ainsi un temps de réponse optimal.

Dans cette section, nous présentons la formation de modèles de versions explorables en parallèle dans la section 4.4.1. Puis, nous proposons une approche pour l'élagage du graphe d'exploration dynamique des versions dans la section 4.4.2. L'exploration des versions candidates d'un modèle de versions est décrite dans la section 4.4.3.

4.4.1 Formation des modèles de versions

La parallélisation de l'exploration de deux modèles de versions est possible s'il n'y a pas de recouvrement entre leurs ensembles respectifs de versions candidates. Deux versions candidates concernent des ensembles d'instances disjoints si elles contiennent respectivement des propriétés qui n'apparaissent jamais ensemble, c'est à dire, deux propriétés faisant partie de la même règle d'exclusion. Par exemple, soit la règle d'exclusion $r_1 = p_3 \mid p_4$; et soient les deux versions candidates $v_1 = \{p_1, p_2, p_3\}$ et $v_2 = \{p_1, p_2, p_4\}$. Les versions v_1 et v_2

ont deux ensembles d'instances complètement disjoints et donc elles ne se recouvrent pas, car elles contiennent respectivement les propriétés p_3 et p_4 qui font partie d'une règle d'exclusion. Nous définissons un modèle de versions comme suit.

Définition (Modèle de versions). Un modèle de versions M est un ensemble de propriétés qui caractérise un ensemble de versions candidates V_M. Soit E l'ensemble réduit des propriétés d'un type t ; l'ensemble M est tel que :

- Toute propriété de M est une propriété de E ;
- Certaines propriétés de M peuvent être obligatoires ; si p est obligatoire dans M alors p doit apparaître dans toutes les versions candidates de V_M. Une propriété obligatoire est notée $\bar{p}$;
- L'ensemble V_M est l'ensemble de toutes les versions candidates possibles générées à partir de M et qui contiennent les propriétés obligatoires de M.

Notons que l'ensemble réduit des propriétés E est un modèle de versions particulier où aucune propriété n'est obligatoire.

Un modèle de versions est un ensemble de propriétés. Il est extrait d'abord à partir de l'ensemble réduit des propriétés E associées à un type. Il peut également être extrait à partir d'un autre modèle de versions. Toutes les propriétés dans E sont optionnelles (non obligatoires) lors de la génération d'une version candidate, c'est à dire, une version candidate générée à partir de E peut ne pas avoir une propriété p qui appartient à E. Cependant, un modèle de versions M_i peut avoir des propriétés obligatoires pour que ses versions candidates ne se recouvrent pas avec les versions candidates d'un autre modèle de versions M_j qui est exploré en parallèle.

Nous présentons dans la section 4.4.1.1 la façon dont une règle d'exclusion est exploitée pour former les modèles de versions à explorer en parallèle. Puis, nous présentons la prise en compte de plusieurs règles d'exclusion dans la section 4.4.1.2.

4.4.1.1 Exploitation d'une règle d'exclusion

Soit un modèle de versions $F = \{p_1, ..., p_i, ..., p_j, ..., p_n\}$ où chaque propriété est également une propriété de l'ensemble réduit de propriétés E ; et soit la règle d'exclusion $r_1 = p_i \mid p_j$. Pour que F soit explorable en parallèle, comme décrit dans la figure 4.8, il faut former trois modèles de versions :

- Un modèle de versions $M_1 = \{p_1, ..., \bar{p_i}, ..., p_n\}$, qui contient toutes les propriétés de F sauf p_j et dans lequel la propriété p_i est obligatoire pour toutes ses versions candidates (notée $\bar{p_i}$) ;
- Un modèle de versions $M_2 = \{p_1, ..., \bar{p_j}, ..., p_n\}$ qui contient toutes les propriétés de F sauf p_i et dans lequel la propriété p_j est obligatoire dans toutes ses versions candidates (notée $\bar{p_j}$) ;

– Un modèle de versions $M_3 = \{p_1, ..., p_n\}$ qui contient toutes les propriétés de F sauf p_i et p_j.

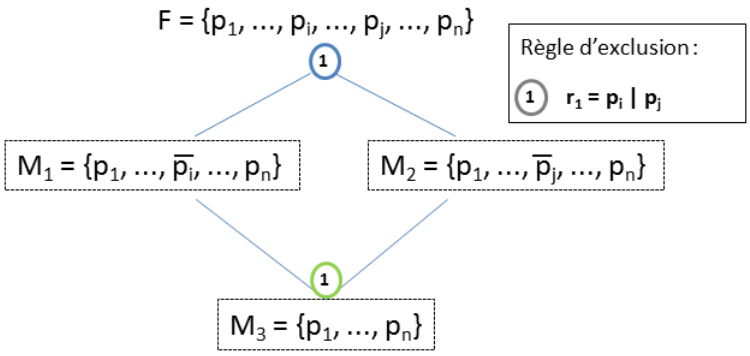


FIGURE 4.8 – Formation de modèles de versions à partir de F guidé par une règle d'exclusion.

Les modèles de versions M_1 et M_2 sont explorables en parallèle car chacun contient une propriété obligatoire inexistante dans l'autre modèle de versions. Par conséquent, il n'y a pas de recouvrement possibles entre les versions candidates des deux modèles de versions. Cependant, le modèle de versions M_3 ne contient pas de propriétés obligatoires issues de la règle d'exclusion r_1, ce qui le rend non exclusif de M_1 et M_2. Par conséquent, des recouvrements sont possibles entre les versions candidates des modèles de versions. Par exemple, soient $v_3 = \{p_1, p_2, p_4\}$ une version candidate issue de M_3, $v_1 = \{p_1, p_2, p_i, p_4\}$ une version candidate issue de M_1, et $v_2 = \{p_1, p_2, p_j, p_4\}$ une version candidate issue de M_2. Quand la source de données est interrogée pour avoir le nombre d'instances ayant les propriétés de v_3, la réponse va inclure les instances ayant les propriétés des versions v_1, v_2 et v_3. Pour obtenir le nombre d'instances ayant uniquement les propriétés de v_3, il faut d'abord interroger la source sur le nombre d'instances ayant les propriétés de v_1, v_2 puis v_3, il faut ensuite soustraire le nombre d'instances ayant les propriétés de v_1 ou v_2 du nombre d'instances ayant les propriétés de v_3, comme dans la formule 4.1. De manière générale, il faut d'abord trouver les versions des modèles de versions M_1 et M_2 en parallèle puis explorer le modèle de versions M_3 comme décrit dans la figure 4.8.

Dans la figure 4.9, supposons le modèle de versions $M_1 = \{p_1, ..., \bar{p_i}, ...p_k, ..., p_n\}$ et la règle d'exclusion $r_2 = p_i|p_k$. Dans ce cas, la règle d'exclusion r_2 est exploitée en supprimant la propriété p_k de M_1 car la propriété $\bar{p_i}$ est obligatoire dans M_1 et par conséquent elle doit figurer dans tous ses sous-ensembles.

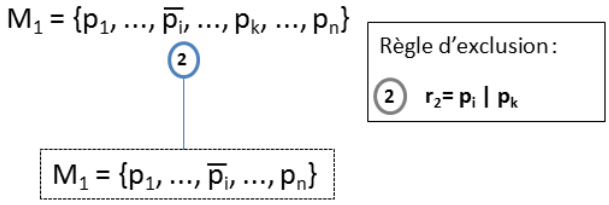


FIGURE 4.9 – Exploitation d'une règle d'exclusion concernant une propriété obligatoire.

4.4.1.2 Prise en compte de plusieurs règles d'exclusion

Les propriétés d'un type peuvent avoir plusieurs règles d'exclusion. Nous proposons d'ordonner ces règles en commençant par celles dont les propriétés ont le plus de contraintes, c'est à dire qui sont concernées par le plus grand nombre de règles d'exclusion, afin de minimiser la taille potentielle de l'arbre d'exploration à parcourir. Soit $R = \{r_1, r_2, ..., r_n\}$ cet ensemble ordonné de règles d'exclusion.

Le nombre de modèles de versions à explorer en parallèle à un instant donné ne doit pas être supérieur à la capacité de la source de données à traiter des requêtes en parallèle. En effet, chaque modèle de versions est exploré par un sous-processus en envoyant séquentiellement des requêtes à la source pour valider ses versions candidates. Si le nombre de sous-processus est supérieur à la capacité de la source de données à traiter des requêtes en parallèle, cela va augmenter le temps de réponse de la source de données. Dans ce cas, il est préférable de ne pas paralléliser certaines tâches d'exploration afin de ne pas surcharger la source de données et garder ainsi un temps de réponse optimal. Pour cela, nous proposons de traiter un ensemble de propriétés selon les règles d'exclusion en veillant à ce que le nombre de modèle de versions ne dépasse pas $MaxTâche$ qui est la capacité de la source de données à traiter des requêtes en parallèle. En effet, il est préférable d'exploiter certaines règles d'exclusion localement par des *sauts*, comme décrit pour l'exploration séquentielle des versions avec *SchemaDecrypt* (voir section 4.3.5), que d'introduire des délais d'attente.

Comme décrit dans l'algorithme 11, les modèles de versions sont générés à partir de E en prenant en compte successivement chaque règle d'exclusion, et ce jusqu'à atteindre la capacité maximale de parallélisation de la source de données. Pour cela, il faut parcourir la liste des règles d'exclusion R tant que le nombre de modèles de versions à explorer en parallèle ($NbThread$) à un instant donné n'a pas atteint la capacité maximale de la source de données à traiter des requêtes en parallèle ($MaxTâche$).

Soit $modèles\_versions$ la liste des modèles de versions de E à traiter en parallèle à un instant donné. Cette liste contient initialement l'ensemble de propriétés E. Pour chaque élément M_r dans la liste $modèles\_versions$ et tant que $NbThread$ est inférieur à $MaxTâche$, nous vérifions si M_r respecte la règle d'ex-

Algorithm 11: Formation dynamique des modèles de versions

Input: la liste des propriétés à combiner E, le profil Pc, la liste des versions validées $liste\_versions$, l'ensemble des règles d'exclusion R, Nombre de tâche parallélisable pour la source de données $MaxTâche$

```
1  j = 1 ; sous_ensembles.add(E) ; NbThread++ ;//initialisé à 1 par le premier thread;
2  while (∃α_i ∈ Pc : α_i! = 0) do
3      for ∀r_i ∈ R, i allant de j à n do
4          if NbThread = MaxTâche then
5              break;
6          end
7          for ∀M_r ∈ liste do
8              if NbThread = MaxTâche) then
9                  break;
10             end
11             if NonRespet(M_r, r_i) then
12                 Remplacer dans modèles_versions l'ensemble M_r par ce que retourne la fonction RendreCompatible(M_r, r_i);
13                 if |RendreCompatible(M_r, r_i)| = 2 then
14                     Enlever de M_r les propriétés de la contrainte r_i;
15                     Empiler(pile^M, M_r) ; Empiler(pile^R, i) ; NbThread++;
16                 end
17             end
18         end
19     end
20     for ∀M_r ∈ liste do
21         Créer un thread T_k;
22         T_k.SchemaDecrypte(M_r) //chaque thread met à jour la liste des versions validées et Pc;
23     end
24     j = Dépiler(pile^R);
25     Attendre que tous les sous-processus (threads) créés par la contrainte j aient fini ou que NbThread = 0 ;
26     à chaque fois qu'un sous-processus fini : NbThread − − ;
27     modèles_versions = ∅ ; M_r = Dépiler(pile^M);
28     Enlever de M_r les propriétés dont la probabilité est à 0 dans Pc;
29     sous_ensembles.add(E_r);
30     while (TeteListe(pile^R) = j) do
31         M_r = Dépiler(pile^M);
32         Enlever de M_r les propriétés dont la probabilité est à 0 dans Pc;
33         sous_ensembles.add(M_r) ; Dépiler(pile^R);
34     end
35     j++ ;
36 end
```

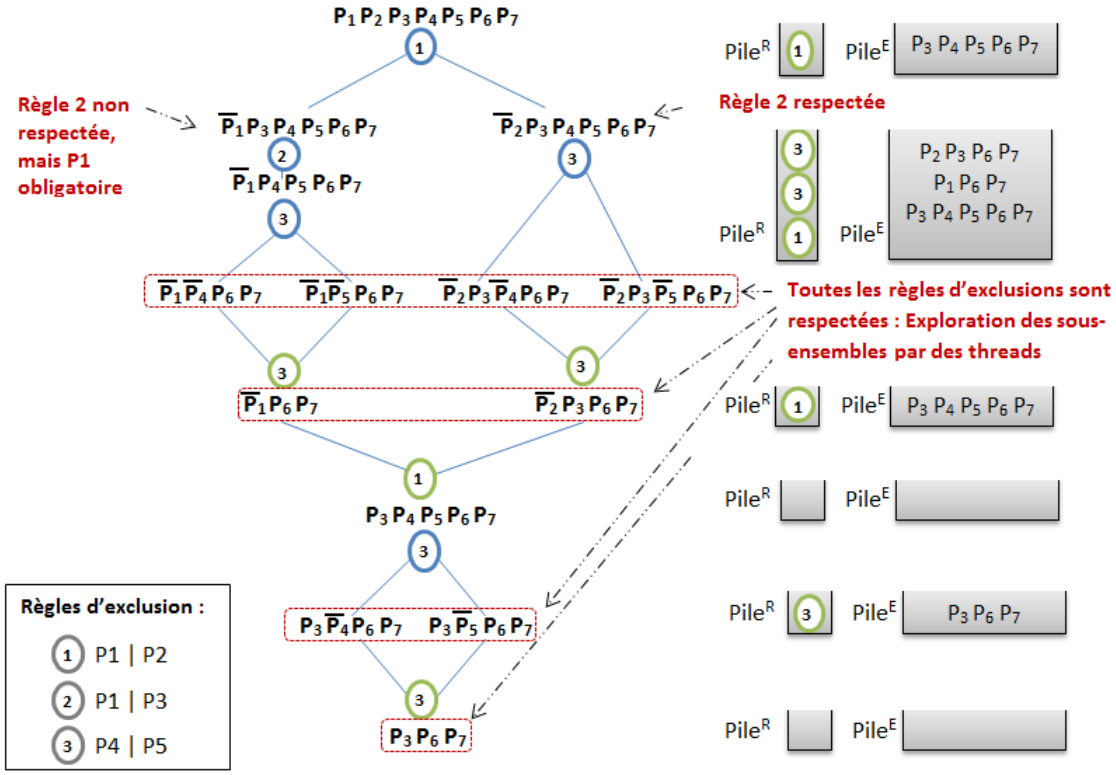


FIGURE 4.10 – Exemple d'un graphe d'exploration dynamique des modèles de versions.

clusion courante r_i. Si ce n'est pas le cas, l'élément M_r est traité en exploitant la règle r_i comme décrit dans la section 4.4.1.1. Le traitement de M_r peut générer deux modèles de versions à traiter en parallèle sauvegardés dans la liste *modèles_versions*, et un modèle de versions à explorer ultérieurement empilé dans $pile^M$. La règle d'exclusion qui a causé l'empilement d'un modèle de versions est également empilée dans $pile^R$ pour pouvoir synchroniser le traitement des modèles de versions empilés. Lorsque le nombre de modèles de versions dans la liste *modèles_versions* atteint $MaxTâche$ ou quand toutes les règles d'exclusion sont traitées, un sous-processus est créé pour chaque modèle de versions dans la liste *modèles_versions* .

Chaque modèle de versions est exploré par un sous-processus. Pour synchroniser la recherche des versions d'un type entre les sous-processus, des données sont partagées entre les sous-processus avec écriture concurrente, comme suit :

- Pc : le profil des probabilités des propriétés du type qui est mis à jour à chaque fois qu'un sous-processus valide une version ;
- *Liste_versions* : la liste des versions validées. A chaque fois qu'un sous-processus valide une version, il l'ajoute à cette liste avec son nombre d'occurrences ;

- *NbThread* : le nombre de tâches d'exploration des versions exécutées en parallèle. Ce nombre est incrémenté à chaque fois qu'un sous-processus est créé pour explorer un modèle de versions, et il est décrémenté quand le sous-processus fini son exploration. *NbThread* ne doit pas dépasser la capacité de la source de données à traiter des requêtes en parallèle $MaxTâche$.

Chaque sous-processus met à jour la liste des versions validées *liste_versions* et le profil du type Pc selon le nombre d'occurrences des versions validées. A chaque fois qu'un sous-processus prend fin, la valeur de *NbThread* est décrémentée. Quand tous les sous-processus créés par la règle courante se terminent ou que $NbThread = 0$, les modèles de versions empilés par le traitement de la règle courante sont traités. La liste des modèles de versions contient cette fois les modèles de versions empilés par le traitement de la contrainte courante, après suppression des propriétés dont la probabilité est à 0 dans le profil du type, ce qui permet de réduire l'espace de recherche de 2^{n-1} pour chaque propriété supprimée, où n représente le nombre de propriétés d'un modèle de versions. Puis, une prochaine itération permettra l'exploitation de la règle suivante tant que toutes les versions du type ne sont pas trouvées. Le processus s'arrête quand toutes les probabilités des propriétés dans le profil du type sont égales à 0.

La figure 4.10 montre un exemple d'exécution de l'algorithme 11. Nous pouvons voir que la formation des modèles de versions et leur plan d'exécution forme un graphe d'exploration dynamique. Ce dernier est construit au fur et à mesure que les sous-processus créés par la même règle terminent leur exploration et tant que toutes les versions du types n'ont pas été trouvées.

La figure 4.11 montre l'exemple de la figure 4.11 en supposant que la source de données peut traiter au plus 3 requêtes en parallèle ($MaxTâche = 3$). Dans ce cas, la liste des règles d'exclusion est exploitée pour la parallélisation tant que le nombre de modèles de versions de E n'a pas atteint $MaxTâche$.

Algorithm 12: Fonction RendreCompatible

Input: la liste des propriétés M_r, la règle d'exclusion r_i

```
1  Soit r_i = p | p' ;
2  if (p et p' ne sont pas fixes dans M_r) then
3  |  Soit M_j = M_r - {p};
4  |  Fixer p' dans M_j ;
5  |  M_r = M_r - {p'};
6  |  Fixer p dans M_r ;
7  |  Retourner {M_r, M_j};
8  else
9  |  Supprimer de M_r la propriété qui n'est pas fixe : p ou p';
10 |  Retourner {M_r}
11 end
```

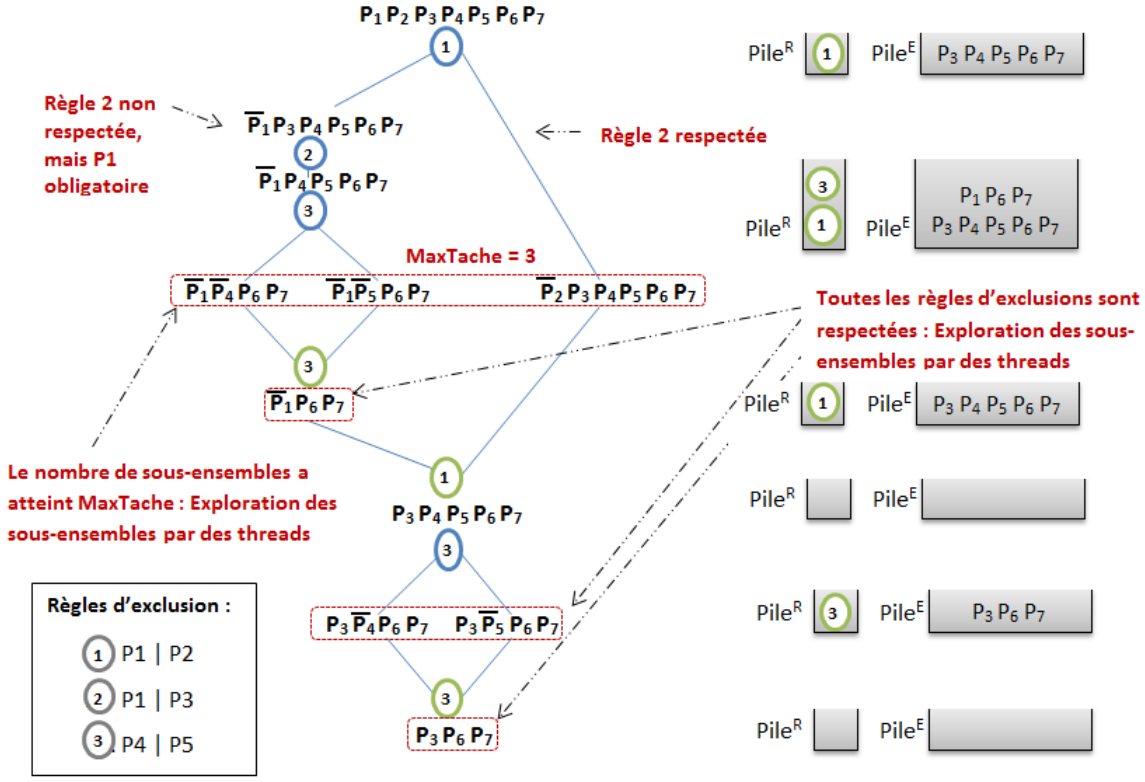


FIGURE 4.11 – Exemple d'un graphe d'exploration dynamique des modèles de versions avec capacité de parallélisation de la source de données limitée à $MaxTâche = 3$.

La fonction *RendreCompatible* décrite dans l'algorithme 12 permet de transformer un ensemble de propriétés M_r qui ne respecte pas une règle d'exclusion r_i en un ou deux sous ensembles de propriétés respectant cette règle d'exclusion comme décrit dans la section 4.4.1.

4.4.2 Élagage du graphe d'exploration dynamique des versions

A la création d'un nouveau modèle de versions dans le graphe d'exploration, il faut vérifier qu'il existe bien des combinaisons avec les propriétés obligatoires de ce nouveau modèle de versions. Cela est fait au moyen d'une requête pour trouver le nombre d'instances décrites par ces propriétés obligatoires. Si cela retourne une valeur égale à 0, alors le modèle de versions est supprimé du graphe d'exploration, ce qui revient à élaguer le graphe d'exploration de la branche correspondante. Ceci permet de réduire le nombre de combinaison à tester de 2^n, où n représente le nombre de propriétés non obligatoires de ce modèle de versions.

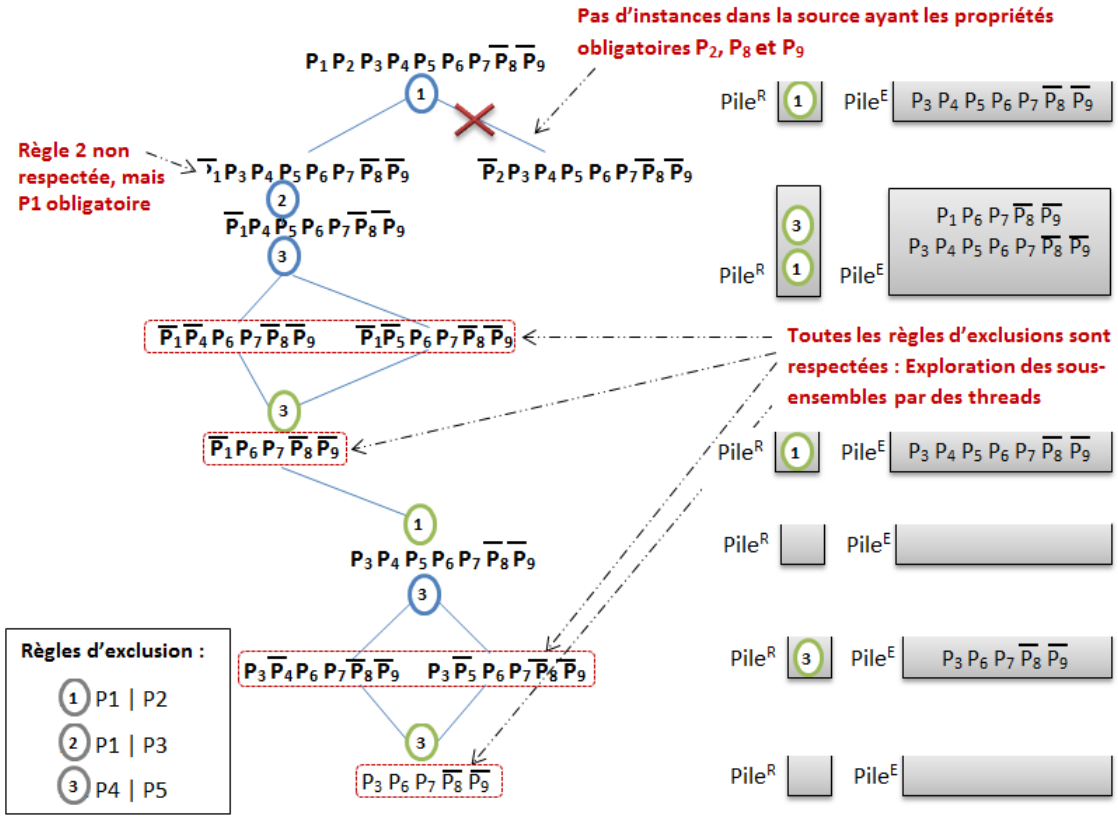


FIGURE 4.12 – Exemple d'un graphe d'exploration dynamique élagué.

La figure 4.12 montre l'exemple de la figure 4.10 modifié de façon à illustrer l'élagage du graphe, en supposant que les propriétés p_8 et p_9 sont obligatoires. Lors de l'exploitation de la règle d'exclusion $r_1 = p_1 \mid p_2$, un modèle de versions est créé, contenant les propriétés obligatoires p_2, p_8 et p_9. La source de données est interrogée pour savoir s'il existe des versions du type contenant les propriétés obligatoires p_2, p_8 et p_9. La source de données renvoie une valeur égale à 0, ce qui indique que ce modèle de versions ne correspond à aucune version, donc sa branche est élaguée dans le graphe d'exploration. Nous pouvons voir que dans la figure 4.12, 6 sous-processus sont exécutés, alors que dans la figure 4.10 précédente, 9 sous-processus sont exécutés.

4.4.3 Exploration d'un modèle de versions

Chaque modèle de versions est exploré par un sous-processus. Rappelons qu'un modèle de versions peut être formé de deux types de propriétés qui sont les suivants :

– Les propriétés obligatoires : elles permettent la parallélisation de l'exploration des différents modèles de versions car elles assurent qu'il n'y a pas

d'intersection ou de recouvrement entre les versions candidates des modèles de versions. Lors de l'exploration d'un modèle de versions par un sous-processus, les propriétés obligatoires sont toujours présentes dans les versions candidates ;
- Les propriétés non obligatoires (optionnelles) : elles forment les différentes combinaisons des versions candidates à tester.

Dans ce qui suit nous allons d'abord montrer dans la section 4.4.3.1 comment l'ensemble des propriétés non obligatoires est réduit, puis la découverte de nouvelles règles d'occurrences qui ne concernent qu'un modèle de versions dans la section 4.4.3.2, et enfin l'exploitation des règles d'occurrences au niveau d'un modèle de versions dans la section 4.4.3.3.

4.4.3.1 Réduction du nombre de propriétés non obligatoires

Nous proposons de réduire le nombre de propriétés non obligatoires en testant chaque propriété non obligatoire individuellement avec les propriétés obligatoires du modèle de versions, pour rechercher des versions valides du type dans la source de données. Si le nombre d'instances du type décrites par l'ensemble formé par les propriétés obligatoires du modèle de versions et l'une des propriétés non obligatoires est égale à 0, alors la propriété non obligatoire est supprimée du modèle de versions à explorer. Ceci permet de réduire le nombre de versions candidates à tester de 2^{n-1} pour chaque propriété non obligatoire supprimée, où n représente le nombre de propriétés non obligatoires au moment de la suppression de la propriété.

4.4.3.2 Découverte de nouvelles règles d'occurrences pour un modèle de versions

Un modèle de versions représente un sous-ensemble d'instances du type dans la source de données. Les instances d'un modèle de versions sont caractérisées par les propriétés obligatoires du modèle. Les règles d'inclusion et d'exclusion qui caractérisent le type sont vérifiées pour les instances correspondant à ce modèle de versions ; mais en plus, d'autres règles d'inclusion et d'exclusion peuvent les caractériser en propre. Ces nouvelles règles sont détectées de la même manière que pour les instances du type, sauf que les propriétés obligatoires du modèle de versions sont ajoutées à chaque requête pour pouvoir distinguer les instances de ce modèle de versions des autres instances du type.

Pour trouver les règles d'exclusion et d'inclusion qui concernent le sous-ensemble d'instances traitées par le sous-processus, il faut d'abord construire le profil $P_{t'}$ du modèle de versions traité. La probabilité de chaque propriété non obligatoire du modèle est calculée, en ajoutant à chaque fois les propriétés obligatoires dans la requête pour que cela ne concerne que le modèle de versions traité par le sous-processus.

Chaque sous-processus trouve les règles d'inclusion et d'exclusion entre les propriétés de son modèle de versions, comme décrit précédemment dans *SchemaDecrypt* (voir section 4.3.3), mais en ajoutant toutes les propriétés obligatoires de son modèle de versions. Soient P_{oblig} l'ensemble des propriétés obligatoires, $\alpha_{i,P_{oblig}}$ (resp. $\alpha_{j,P_{oblig}}$) la probabilité d'une propriété p_i (resp. p_j) de décrire une instance d'un sous-ensemble t' des instances d'un type t, et $\alpha_{i,j,P_{oblig}}$ la probabilité des propriétés p_i et p_j de décrire une instance du sous-ensemble t'. Nous proposons de trouver les règles d'inclusion et d'exclusion entre les propriétés du modèle de versions M comme suit :

Règle d'inclusion. Les règles d'inclusion entre les propriétés du modèle de versions M sont déterminées en testant chaque paire de propriétés p_i, $p_j \in M$, avec $\alpha_{P_{oblig},i} != \alpha_{P_{oblig},j}$ dans le profil du sous-ensemble d'instances correspondant t', comme suit :

- Si $(\alpha_{i,j,P_{oblig}} = \alpha_i, P_{oblig})$ alors $(p_i \Rightarrow p_j$ dans $t')$
- Si $(\alpha_{i,j,P_{oblig}} = \alpha_j, P_{oblig})$ alors $(p_j \Rightarrow p_i$ dans $t')$

Règle d'exclusion. Les règles d'exclusion entre les propriétés du modèle de versions M sont déterminées en testant chaque paire de propriétés p_i, $p_j \in M$, avec $(\alpha_{i,P_{oblig}} + \alpha_{j,P_{oblig}} <= 1)$ dans le profil du sous-ensemble d'instances correspondant t', comme suit :

- Si $(\alpha_{i,j,P_{oblig}} = 0)$ alors $(p_{i,P_{oblig}} \mid p_{j,P_{oblig}}$ dans $t')$

Ces nouvelles règles d'exclusion sont utilisées pour paralléliser la découverte des versions comme décrit précédemment jusqu'à ce que la capacité maximale de parallélisation de la source $MaxTâche$ soit atteinte. Ensuite, les règles d'exclusion et d'inclusion propres à chaque modèle de versions sont exploitées avec des *sauts* comme décrit dans l'exploration séquentielle des versions avec *SchemaDecrypt* en section 4.3.5.

4.4.3.3 Exploitation des règles d'occurrences lors du traitement d'un modèle de versions

Les règles d'exclusion sont exploitées initialement pour trouver le graphe d'exploration qui forme les modèles de versions. Cependant, si le nombre de modèles de versions atteint le nombre maximum de requêtes que la source de données peut traiter en parallèle $MaxTâche$, comme dans la figure 4.11, certaines règles restent inexploitées dans certains modèle de versions, comme la règle $r_3 = p_4 \mid p_5$ dans le modèle de versions $M_2 = \{\bar{p_2}, p_3, p_4, p_5, p_6, p_7\}$. Dans ce cas, ces règles sont exploitées par des *sauts* comme décrit précédemment dans *SchemaDecrypt* (voir section 4.3.5).

Chaque processus d'exploration d'un modèle de versions exploite ses règles d'inclusion par des *sauts* et en réduisant le nombre de propriétés dans une re-

quête comme décrit précédemment dans la section 4.3.5. De plus certaines règles d'inclusion sont exploitées de manière spécifique en raison de la présence de propriétés obligatoires dans le modèle de versions et parce que certaines de ces règles concernent uniquement ce modèle de versions. L'exploitation de ces règles pour un modèle de versions M se fait comme suit :

1. S'il existe une règle d'inclusion $\bar{p}_i \Rightarrow p_j$, cela signifie que la propriété non obligatoire p_j est toujours présente dans les versions découvertes du modèle. Dans ce cas, il n'est pas utile de la tester et elle est supprimée de M et ajoutée directement dans chaque version validée ;
2. Si toutes les versions contenant une propriété non obligatoire p_j ont été trouvées, ce qui correspond à l'annulation de la probabilité de p_j dans le profil de type, alors p_j et toutes les propriétés p_i telles que : $p_i \Rightarrow p_j$ sont supprimées de tous les modèles de versions M. Notons que ce cas n'est possible que si cette règle est spécifique à M. En effet, si elle concernait l'ensemble des instances du type, l'annulation de la probabilité de p_j n'aurait lieu que si la probabilité de p_i était également à 0 ;
3. Si la propriété non obligatoire ayant la probabilité la plus élevée est p_j, et si toutes les versions contenant cette propriété sont explorées pour M, alors toutes les propriétés p_i telles que : $p_i \Rightarrow p_j$ sont supprimées de M. Notons que ce cas n'est possible que si cette règle est spécifique à M. En effet, si elle concernait l'ensemble des instances du type, il ne serait pas possible d'avoir fini d'explorer les versions contenant p_j sans avoir au préalable exploré celles contenant p_i.

De manière plus générale, les règles d'inclusion sont exploitées dans M en supprimant les propriétés non obligatoires p_j, dans les deux cas suivants :

- p_i est une propriété obligatoire et $\bar{p}_i \Rightarrow p_j$ (p_j est alors ajoutée directement dans chaque version validée dans M) ;
- A la suppression d'une propriété non obligatoire p_k, et $p_j \Rightarrow p_k$ est une règle d'inclusion spécifique à M.

La suppression de chaque propriété non obligatoire du modèle de versions permet de réduire le nombre de version candidates à tester de 2^{n-1}, où n représente le nombre de propriétés non obligatoires du modèle de versions au moment de la suppression de la propriété. Notons que malgré la suppression d'une propriété de la requête ou de l'ensemble des propriétés non obligatoires, sa probabilité dans le profil est mise à jour pour chaque version trouvée où elle apparaît.

4.5 Analyse du coût de la découverte des versions

Dans cette section, nous discutons le coût de l'approche proposée. Chaque règle d'exclusion permet de paralléliser l'exploration des versions candidates. Cependant, pour qu'une exploration parallèle permette réellement d'améliorer les

performances, il faut que la source de données soit capable de traiter en parallèle plusieurs requêtes. Dans cette analyse du coût de notre approche, nous considérons le cas le plus défavorable où la source ne peut traiter qu'une requête à la fois.

Le nombre de requêtes envoyées à une source de données reflète le coût de l'approche de découverte des versions d'un type en ligne. En effet, le temps de réponse à une requête est le temps le plus important dans le processus de découverte des versions. *SchemaDecrypt* effectue une analyse statistique des propriétés d'un type avant de tenter de découvrir ses versions. Cela entraîne la création d'un profil de type probabiliste en envoyant n requêtes à la source de données, où n est le nombre de propriétés associées à un type. *SchemaDecrypt* calcule également les probabilités entre certaines paires de propriétés pour déduire des ensembles de propriétés co-occurrentes et identifier les règles d'inclusion et d'exclusion. Toutes les paires de propriétés ne sont pas testées, mais seulement celles qui satisfont certaines conditions, comme décrit dans les sections 4.3.2 et 4.3.3. Par conséquent, *SchemaDecrypt* interroge la source de données au plus $C_n^2 = n \times (n-1)/2$ fois, ce qui représente le calcul de probabilité de toutes les paires de propriétés parmi n.

L'**algorithme 9** représente la tâche la plus coûteuse de *SchemaDecrypt* : la génération dynamique des versions candidates. En effet, le temps de traitement d'une requête testant une version candidate est plus important que le temps nécessaire pour détecter une règle : une requête permettant de tester une version candidate est composée de toutes les propriétés de la version candidate alors qu'une requête pour détecter une règle est composée de seulement deux propriétés. Notons que le temps de réponse à une requête augmente à mesure que le nombre de propriétés dans la requête augmente. Pour évaluer la complexité de notre approche, il est donc nécessaire d'estimer *a priori* le nombre de requêtes envoyées à la source de données pour trouver les versions.

Nous ne pouvons pas déterminer *a priori* le nombre de requêtes générées par *SchemaDecrypt* pour être envoyées à la source de données dans l'algorithme 9, car elles sont envoyées jusqu'à ce que toutes les versions du type soient trouvées ; et nous ne pouvons pas déterminer le nombre de versions d'un type *a priori*. Cependant, nous pouvons estimer le nombre de requêtes envoyées à la source de données dans le cas le plus défavorable : les versions du type ne sont pas trouvées tant que toutes les versions candidates potentielles ne sont pas testées par une requête.

Le nombre de requêtes envoyées à la source de données par *SchemaDecrypt* est lié au nombre de propriétés associées à un type et au nombre de règles d'inclusion et d'exclusion découvertes. Soit n le nombre de propriétés associées à un type dans l'ensemble E, la complexité de la recherche exhaustive des versions du type est donc de 2^n, ce qui représente le nombre de versions candidates.

Soit h le nombre de règles d'inclusion et d'exclusion découvertes pour les entités du type. Chaque règle diminue la complexité d'une recherche exhaustive (2^n) de 2^{n-2}, car chaque règle implique deux propriétés. Par exemple, pour une règle r = $p_i \mid p_j$, le nombre de combinaisons à ne pas tester est de 2^{n-2}. Le nombre de combinaisons à tester par une requête est de 2^n - 2^{n-2}. Les combinaisons à ne pas tester sont les suivantes :

p_n .....p_j.....p_i.....p_1	
XXXXX1XXXX1.....X	avec X ∈ {0, 1}

Chaque règle diminue la complexité d'une recherche exhaustive de 2^{n-2}, car chaque règle implique deux propriétés. Cependant, une règle r_1 peut rentrer en collision avec une autre règle r_2. Nous considérons qu'une collision entre les règles r_1 et r_2 se produit lorsque la version candidate ne respecte pas les deux règles r_1 et r_2. Dans ce cas, la réduction du nombre de combinaisons n'est pas de $2 * 2^{n-2}$, mais $2 * 2^{n-2}$ - le nombre de collisions. Plus généralement, soit $NbCollision$ le nombre de collisions entre les règles, le nombre de combinaisons $combNb$ à tester en fonction du nombre de règles pour un type est calculé comme dans la formule 4.5.

$$combNb = 2^n - h * 2^{n-2} + NbCollision \tag{4.5}$$

Par exemple, considérons les deux règles : $r_1 = p_i \Rightarrow p_j$ et la règle $r_2 = p_i \mid p_k$; les combinaisons rejetées sont les suivantes :

p_n.....p_k.....p_j.....p_i.....p_1	
XXXXX0XXXX0XXXX1XXXXX	r_1 non respectée
XXXXX1XXXX0XXXX1XXXXX	r_1 et r_2 non respectées => NbCollision = 2^{n-3}
XXXXX1XXXX1XXXX1XXXXX	r_2 non respectée
avec X ∈ {0, 1}	

Dans cet exemple, le nombre de collisions est 2^{n-3}, car les deux règles impliquent trois propriétés. Deux règles sont indépendantes si elles ne partagent pas de propriétés en portant sur quatre propriétés distinctes. Dans ce cas, le nombre de collisions peut être de 2^{n-4}. Par exemple, considérons la règle $r_1 = p_i \mid p_j$ et la règle $r_2 = p_k \Rightarrow p_v$; les combinaisons écartées sont les suivantes :

p_n .....p_v.....p_k..... p_j.....p_i.....p_1	avec X ∈ {0, 1}
XXXXX0XXXX0XXXXX1XXXX1XXXXX	r_1 non respectée
XXXXX0XXXX1XXXXX0XXXX0XXXXX	r_2 non respectée
XXXXX0XXXX1XXXXX0XXXX1XXXXX	r_2 non respectée
XXXXX0XXXX1XXXXX1XXXX0XXXXX	r_2 non respectée
XXXXX0XXXX1XXXXX1XXXX1XXXXX	r_1 et r_2 non respectée => NbCollision = 2^{n-4}
XXXXX1XXXX0XXXXX1XXXX1XXXXX	r_1 non respectée
XXXXX1XXXX1XXXXX1XXXX1XXXXX	r_1 non respectée

Dans certains cas, deux règles ne rentrent jamais en collision. Par exemple, considérons les règles $r_1 = p_i \Rightarrow p_j$ et $r_2 = p_j \mid p_k$. Une collision se produit quand :

- r_1 est violée : si $bit(p_i) = 1$ et $bit(p_j) = 0$
- r_2 est violée : si $bit(p_j) = 1$ et $bit(p_k) = 1$

La collision est impossible car $bit(p_j)$ ne peut pas être égal à 1 et à 0 en même temps, donc le nombre de collisions entre les règles r_1 et r_2 est de 0.

Nous pouvons constater que le nombre de collisions entre les règles diffère selon les types de règles et leur dépendance. Nous résumons les différents cas pour déterminer le nombre de collisions entre deux règles $NbCollision(r_a, r_b)$ dans la Formule 9.

$$collisionNb(r_a, r_b) = \begin{cases} 2^{n-4} & \text{si } r_a = p_i \varphi p_j \text{ et } r_b = p_k \varphi p_v, \text{ avec } \varphi \in \{\Rightarrow, \mid\} \\ 2^{n-3} & \text{si } \begin{cases} r_a = p_i \Rightarrow p_j \text{ et } r_b = p_i \Rightarrow p_k \\ r_a = p_i \Rightarrow p_j \text{ et } r_b = p_k \Rightarrow p_j \\ r_a = p_i \mid p_j \text{ et } r_b = p_j \mid p_k \\ r_a = p_i \Rightarrow p_j \text{ et } r_b = p_i \mid p_k \end{cases} \\ 0 & \text{si } \begin{cases} r_a = p_i \Rightarrow p_j \text{ et } r_b = p_j \Rightarrow p_k \\ r_a = p_i \Rightarrow p_j \text{ et } r_b = p_k \Rightarrow p_i \\ r_a = p_i \Rightarrow p_j \text{ et } r_b = p_j \mid p_k \end{cases} \end{cases} \quad (9)$$

avec p_i, p_j, p_k, p_v quatre propriétés distinctes

Le nombre de requêtes envoyées pour découvrir les versions avec *SchemaDecrypt* dépend du nombre de règles découvertes. Soit h le nombre de règles d'occurrences et n le nombre de propriétés associées à un type. Chaque règle diminue la complexité d'une recherche exhaustive (2^n) de 2^{n-2}, car chaque règle implique deux propriétés. Cependant, certaines versions candidates ne res-

pectent pas plusieurs règles à la fois, ce qui entraîne une collision entre les règles. Soit $NbCollision(r_a, r_b)$ le nombre de collisions entre deux règles r_a, r_b tel que décrit dans la formule 9. Soit *NbQueries* le nombre de requêtes envoyées à la source de données pour tester les versions candidates en fonction du nombre de collisions *TotalColl* entre chaque paire de règles. Le nombre de requêtes envoyées pour découvrir les versions avec *SchemaDecrypt* est calculé dans le cas le plus défavorable comme le montre la formule 4.6.

$$NbQueries(n, h) = 2^n - h * 2^{n-2} + TotalColl \tag{4.6}$$

La formule 4.7 donne le nombre de collisions entre chaque paire de règles comme suit :

$$TotalColl = \sum_{a=2}^{h} \sum_{b=1}^{a-1} NbCollision(r_a, r_b) \tag{4.7}$$

4.6 Évaluation

Cette section présente quelques résultats d'expérimentation en utilisant l'approche *SchemaDecrypt* pour trouver les différentes versions d'un type. Nous avons également évalué les performances de *SchemaDecrypt++* et nous les avons comparées à celles de *SchemaDecrypt*, afin de montrer l'effet du parallélisme et de l'élagage dynamique du graphe d'exploration sur une source de données réelle.

4.6.1 Source de données et méthodologie

Nous avons évalué les performances de notre approche pour fournir les versions exactes d'un type, en utilisant une source de données distante réelle : *DBpedia*[3], qui contient actuellement plus de 3.64 millions d'instances ; elle est composée de plus de 1.2 milliard de triplets RDF extraits de *Wikipédia*. Nous avons choisi cette source de données car le nombre de propriétés d'une instance est plutôt élevé : 150 propriétés en moyenne. Pour découvrir les versions d'un type, *SchemaDecrypt* interroge cette source de données distante via un point d'accès SPARQL[4].

Nous avons utilisé *SchemaDecrypt*(++) pour découvrir les versions des types suivants dans la source de données *DBpedia* : *Poet*, *Restaurant*, *Museum*, *ShoppingMall* et *Park*. Nous avons délibérément choisi des types avec un nombre élevé de propriétés (entre 269 et 473) pour tester l'efficacité de l'approche. Notons que *DBpedia* est en constante évolution car elle est automatiquement enrichie à partir de *Wikipédia*. Par conséquent, il peut y avoir une légère différence dans le nombre

3. DBpedia : dbpedia.org
4. DBpedia, point d'accès SPARQL : http ://dbpedia.org/sparql

de propriétés d'un type à différentes périodes. Cela motive encore une fois l'intérêt d'une approche qui découvre les versions d'un schéma en fonction du contenu courant de la source de données.

SchemaDecrypt + + est implémenté en Java avec une programmation *multi-thread*. Un *thread* est lancé pour chaque processus d'exploration d'un modèle de versions. Pour synchroniser la recherche des versions d'un type entre les *threads*, les ressources suivantes sont partagées en écriture concurrente et en lecture : (i) le profil de type TP qui est mis à jour à chaque fois qu'un *thread* valide une version ; (ii) la liste des versions validées $Liste\_versions$ qui sert entre autre à vérifier les recouvrement entre les versions, à chaque fois qu'un *thread* valide une version, il l'ajoute à cette liste avec son nombre d'occurrences et (iii) le nombre de *threads* lancé en parallèle $NbThread$, à chaque fois qu'un *thread* est créé, il incrémente sa valeur et il la décrémente quand il termine l'exploration de son modèle de versions.

SchemaDecrypt et *SchemaDecrypt* + + sont disponibles en ligne [5]. Nous avons effectué notre expérimentation le 16 Mars 2017, avec une bande passante de 2.4 GHz, sur un ordinateur de bureau : Intel (R) Xeon (R), CPU de 2.80 GHz, 64 bits avec 4 Go de RAM.

4.6.2 Résultats

La figure 4.13 représente les performances de *SchemaDecrypt* en termes de temps de traitement et du nombre de versions candidates testées par une requête, en fonction du nombre de propriétés d'un type, du nombre de règles découvertes et du nombre de versions validées qui représente les versions réelles d'un type.

SchemaDecrypt permet de découvrir les différentes versions d'un type même lorsque le nombre de propriétés est très élevé, par exemple pour le type *Poet* qui possède 473 propriétés. Comme la figure 4.13 (a) le montre, *SchemaDecrypt* parvient à découvrir les différentes versions des types en quelques secondes. Ce résultat est dû à deux idées principales dans *SchemaDecrypt* : (i) l'utilisation du profil de type et l'ordonnancement des propriétés en fonction de leurs probabilités, ce qui conduit à tester les combinaisons les plus probables en premier et à converger rapidement ; (ii) l'exploitation des règles d'occurrences pour éliminer certaines combinaisons et faire des sauts dans le $code\_version$, ce qui réduit considérablement l'espace de recherche.

Le temps de traitement pour découvrir les versions d'un type (voir la figure 4.13 (a)) est proportionnel au nombre de requêtes envoyées à la source de données (voir la figure 4.13 (b)). En effet, le temps de réponse de la requête représente la partie la plus importante du temps total de traitement dans l'**algorithme 9**.

SchemaDecrypt est influencé par le nombre de versions d'un type qui est représenté par le nombre de versions validées dans la figure 4.13 (b). Le temps

5. SchemaDecrypt : http ://github.com/Kenza-Kellou-Menouer/SchemaDecrypt

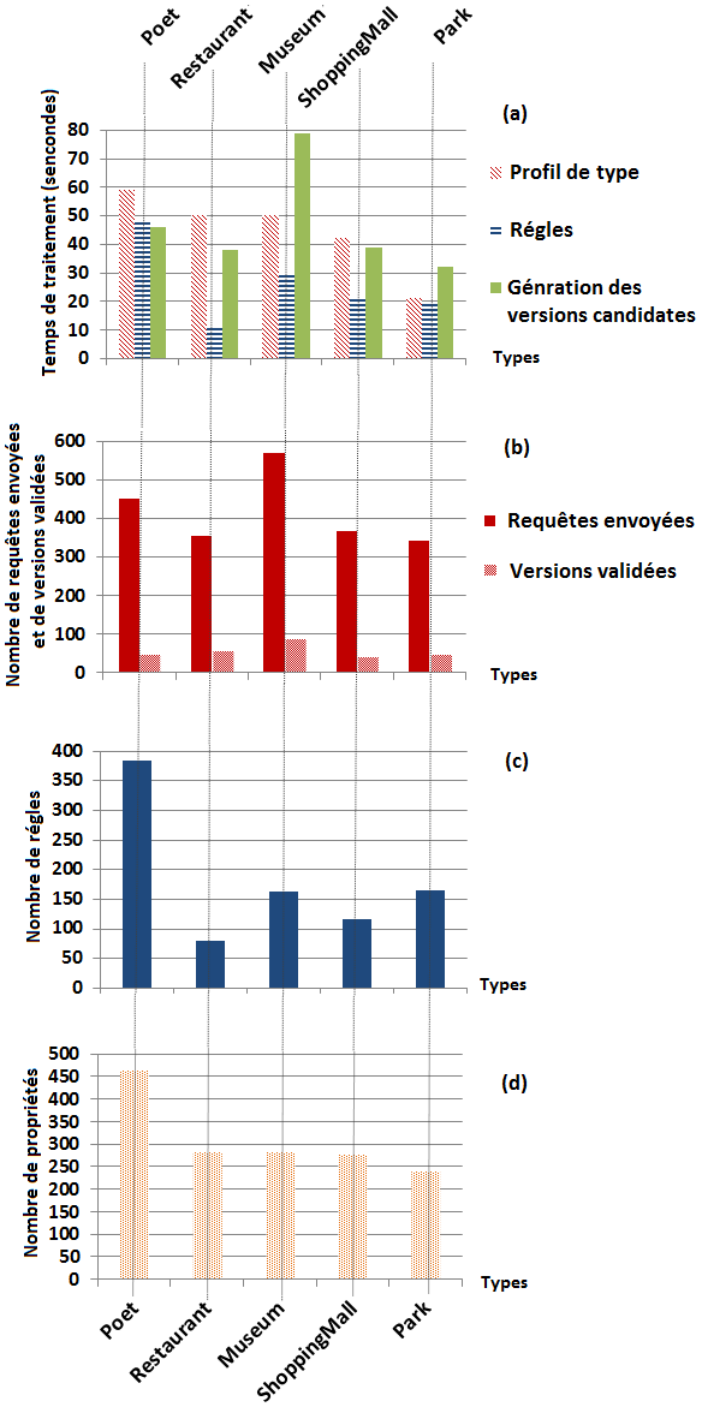


FIGURE 4.13 – Temps de traitement (a) de *SchemaDecrypt* selon : (b) le nombre de versions et de requêtes, (c) le nombre de règles d'occurrences et (d) le nombre de propriétés de chaque type.

nécessaire pour trouver les versions d'un type est proportionnel au nombre de ses versions : plus un type a de versions, plus il faudra à *SchemaDecrypt* de temps pour les retrouver toutes. En effet, le temps nécessaire pour atteindre le critère d'arrêt est proportionnel au nombre de versions car les probabilités des propriétés du profil de type diminuent plus lentement. Par exemple, les types *Restaurant* et *Museum* ont exactement le même nombre de propriétés (291) ; mais le temps nécessaire pour générer les versions candidates pour le type *Restaurant* est de 38 secondes avec 56 versions validées. Pour le type *Museum*, le temps de traitement est de 79 secondes avec 85 versions validées.

Le temps d'exécution est également influencé par le nombre de règles extraites pour un type (voir la figure 4.13 (c)), comme l'illustre le type *Poet* avec 473 propriétés et 386 règles déduites, où le temps de génération des versions candidates est de 46 secondes.

TABLE 4.1 – Les performances de l'approche de base, de *SchemaDecrypt* et de *SchemaDecrypt* + +.

Types	Nombre de versions candidates			Temps de traitement			Versions validées
	Approche de base	Schema Decrypt	Schema Decrypt++	Approche de base*	Schema Decrypt	Schema Decrypt++	
Historian	2^{473}	19738 $\approx$ $2^{14,26}$	330 $\approx 2^{8,36}$	2^{469} sec $\approx$ 2^{444} années $\approx 10^{73,2}$ années	24692 sec (6.85 h)	67 sec	72
Poet	2^{473}	452 $\approx$ $2^{8,82}$	120 $\approx 2^{6,9}$	2^{469} sec $\approx$ 2^{444} années $\approx 10^{73,2}$ années	152 sec	116	44
Restaurant	2^{291}	355 $\approx$ $2^{8,47}$	103 $\approx 2^{6,6}$	2^{287} sec $\approx$ 2^{262} années $\approx 10^{78,5}$ années	106 sec	98 sec	56
Museum	2^{291}	571 $\approx$ $2^{9,15}$	156 $\approx 2^{7,28}$	2^{287} sec $\approx$ 2^{262} années $\approx 10^{78,5}$ années	159 sec	142 sec	85
ShoppingMall	2^{287}	366 $\approx$ $2^{8,51}$	56 $\approx 2^{5,8}$	2^{283} sec $\approx$ 2^{258} années $\approx 10^{77,4}$ années	102 sec	85	38
Stadium	2^{274}	8004 $\approx$ $2^{12,96}$	803 $\approx 2^{9,64}$	2^{270} sec $\approx$ 2^{245} années $\approx 10^{73,5}$ années	1828 sec (30,5 mn)	170 sec (2.83 mn)	220
Mountain	2^{269}	92510 $\approx$ $2^{16,49}$	2113 $\approx$ $2^{11,04}$	2^{265} sec $\approx$ 2^{240} années $\approx 10^{72}$ années	143395 sec (39.8 h)	333 sec (5.55 mn)	376
Park	2^{240}	341 $\approx$ $2^{8,41}$	80 $\approx 2^{5,64}$	2^{236} sec $\approx$ 2^{211} années $\approx 10^{63,3}$ années	80 sec	59 sec	44

*1 année = 31 536 000 sec $\approx 2^{25}$ sec ; et $2^n = 10^{n \times log(2)}$

Dans le tableau 4.1, nous récapitulons les performances de *SchemaDecrypt*, nous présentons les performances de *SchemaDecrypt* + + et nous montrons une estimation des performances de l'approche de base de la recherche de versions de types à partir d'une source de données distante. L'approche de base consiste en une recherche exhaustive des versions comme décrit dans la section 4.2.2. Nous avons testé l'approche de base dans le but d'estimer le temps nécessaire à retrouver les premières versions ; mais cela n'a permis de découvrir aucune version, car des exceptions *timeout* sont survenues pour les requêtes envoyées au serveur Web.

Nous avons tout de même fait une estimation empirique des performances de l'approche de base dans le tableau 4.1. Le nombre de requêtes envoyées à la source par l'approche de base est de 2^n, avec n le nombre de propriétés associées à un type. Nous avons estimé le temps d'exécution de l'approche de base comme suit : comme la source de données *DBpedia* peut tester un maximum de 15 requêtes par seconde [83], le meilleur temps estimé pour le traitement de 2^n requêtes, est donc de $2^n/15$ secondes $\approx 2^{n-4}$ secondes. Le temps de traitement considéré pour *SchemaDecrypt*(++), présenté dans le tableau 4.1, comprend toutes les étapes de l'approche : la construction du profil de type, la déduction des règles et la génération dynamique des versions candidates, comme présenté dans la figure 4.13 (a).

Le tableau 4.2 récapitule les caractéristiques des types selon leur nombre de propriétés, le nombre de règles d'inclusion et d'exclusion découvertes et le nombre de versions. Le nombre de propriétés de chaque type est très élevé, variant de 240 pour le type *Park*, à 473 pour les types *Historian* et *Poet*. Nous pouvons également constater que le nombre de propriétés d'un type, le nombre de règles et le nombre de versions ne sont pas liés.

TABLE 4.2 – Caractéristiques des types.

Types	Propriétés	Règles d'inclusion	Règles d'exclusion	Nombre de versions
Historian	473	79	494	72
Poet	473	98	288	44
Restaurant	291	11	67	56
Museum	291	24	141	85
ShoppingMall	287	29	88	38
Stadium	274	15	34	220
Mountain	269	44	192	376
Park	240	40	126	44

Les résultats du tableau 4.1 montrent que l'approche de base n'est pas réaliste, même sans restrictions du serveur Web. Trouver les versions de ces types, en

utilisant l'approche de base, prend plus d'un milliard de fois l'âge de l'univers. Cependant, *SchemaDecrypt* réussit à découvrir les différentes versions des types en quelques secondes. La justification de ces résultats dans le domaine de la cryptanalyse est simple : l'approche de base est une attaque par force brute qui fait une recherche exhaustive des versions d'un type ; tandis que *SchemaDecrypt* est une attaque probabiliste guidée par le profil du type qui permet de tester les versions les plus probables d'abord et donc d'atteindre rapidement le critère d'arrêt, en plus des règles entre les propriétés qui permettent de réduire l'espace de recherche.

Le temps d'exécution de *SchemaDecrypt* + + est toujours inférieur au temps d'exécution de *SchemaDecrypt*. L'écart entre les deux temps d'exécution se creuse encore plus quand le temps d'exécution de *SchemaDecrypt* est important, comme pour le type *Mountain* avec un temps d'exécution pour *SchemaDecrypt* de 39, 8 heures et un temps d'exécution pour *SchemaDecrypt* + + de 430 fois moins important avec 5, 55 minutes ; et égalemnt pour le type *Historian* avec un temps d'exécution pour *SchemaDecrypt* de 6, 85 heures et un temps d'exécution pour *SchemaDecrypt* + + de 368 fois moins important avec 67 secondes. Quand le temps de traitement de *SchemaDecrypt* est de l'ordre de quelques secondes, la différence entre le temps de traitement avec *SchemaDecrypt* + + n'est pas très visible, néanmoins, le temps de traitement avec *SchemaDecrypt* + + reste toujours inférieur.

Le nombre de versions candidates générées par *SchemaDecrypt* + + est toujours inférieur au nombre de versions candidates générées par *SchemaDecrypt*. Par exemple pour le type *Historian*, *SchemaDecrypt* + + génère 330 versions candidates pour en valider 72, alors que *SchemaDecrypt*, en génère 60 fois plus, (19738 versions candidates) pour trouver le même résultat.

Le nombre des règles d'exclusion permet d'accélérer nettement l'exécution de *SchemaDecrypt*++ sur tous les types et spécialement sur le type *Historian*. Ceci est dû au fait que plus le nombre de règles d'exclusion entre les propriétés d'un type est élevé, plus le nombre de modèles de versions générés sera élevé. L'exploration des modèles de versions en parallèle permet non seulement d'accélérer le temps de traitement, mais aussi de réduire le nombre de versions candidates, car dans chaque modèle de versions, les premières versions générées sont les plus probables.

4.7 Conclusion

Nous avons proposé une approche en ligne pour découvrir le schéma versionné d'une source distante de données massives, sans avoir à télécharger ou à parcourir les données. Pour trouver les différentes versions d'un type, nous proposons de créer un profil de type probabiliste pour guider l'exploration des versions candidates. Nous réduisons le nombre de versions candidates en découvrant des

règles d'inclusion et d'exclusion entre les propriétés d'un type. Nous avons présenté quelques évaluations sur *DBpedia*, qui est accessible via un point d'accès SPARQL. L'approche découvre les versions exactes d'un type avec de bonnes performances, montrant que notre approche est un outil puissant pour comprendre la structure cachée des données du Web sémantique.

Nous avons également proposé une exploration parallèle des versions avec *SchemaDecrypt* + + qui améliore considérablement sa performance. En effet, la présence de règles d'exclusion permet de former des modèles de versions explorables en parallèle. Un modèle de versions doit être conforme aux règles d'exclusion, et contenir toutes les propriétés du type, de sorte qu'il n'y a pas de règles d'exclusion qui les impliquent. L'exploration parallèle des versions avec *SchemaDecrypt*++ optimise considérablement le nombre de versions candidates en explorant chaque modèle de versions en parallèle, car aucune inclusion entre les versions explorées n'est possible.

Les approches de découverte de patterns structurels présentées dans l'état de l'art ne peuvent pas traiter une source de données distante car elles procèdent en parcourant les données. Tandis que *SchemaDecrypt* est capable de découvrir les versions des types d'une source de données distantes sans avoir à la charger en local. La difficulté de cette tâche réside essentiellement sur le fait qu'on ne peut pas parcourir les données, le seul accès est à travers des requêtes pour les interroger. En plus du fait des restrictions d'accès par le serveur comme des *timeout* sur l'exécution d'une requête et la limitation du nombre de requêtes envoyées pour ne pas engorger le réseaux.

Notez que notre approche pourrait également être utilisée pour une source de données locale. Même s'il n'y a pas de restrictions sur la source de données, l'approche permet d'optimiser le temps nécessaire pour la découverte des versions. En effet, la réduction du nombre de propriétés dans une requête permet non seulement d'éviter l'exception *timeout* du serveur mais également de réduire considérablement le temps nécessaire pour répondre à une requête. Les *sauts* au niveau des versions et l'élagage de l'arbre d'exploration permettent non seulement de réduire le nombre de requêtes envoyées au serveur afin de ne pas perdre la priorité de traitement, mais également de réduire le temps nécessaire pour découvrir les versions d'un type. L'exploration des modèles de versions en parallèle permet non seulement d'accélérer le temps de traitement de manière considérable mais aussi de retrouver les versions d'un type en testant moins de versions candidates, car dans chaque modèle de versions, les premières versions générées sont les plus probables, ce qui permet de converger plus rapidement.

Chapitre 5

Annotation des types à l'aide de bases de connaissances disponibles sur le Web

5.1 Introduction

Nous avons proposé dans le chapitre 3, une approche automatique qui permet de découvrir le schéma implicite d'une source de données. Le schéma découvert consiste en un ensemble de clusters d'entités représentant des types (classes), et des liens entre eux. Un problème qui se pose une fois les clusters formés, est de rendre compte de leur sémantique : quel est le sens de chaque groupe d'entités ? Pour trouver le sens de chaque groupe, nous proposons de les annoter. L'annotation consiste à trouver, à partir de bases de connaissances disponibles sur le Web, un ensemble de termes (annotations) pondérés selon leur pertinence et qui expliquent ce que contient un groupe (cluster/classe/type) d'entités.

Dans le cas où le schéma est fourni explicitement dans le jeu de données, l'annotation permettrait également de mieux expliquer des types qui ont déjà un nom en les enrichissant par des annotations complémentaires. En effet, même si le type est défini et a un nom, celui-ci peut ne pas refléter tout le sens de ses instances, par exemple, un type nommé *Person* peut avoir des instances représentant des habitants de certaines villes ou des acheteurs de certains produits. L'annotation permet dans ce cas de compléter la description d'un type en fournissant des informations cachées.

Découvrir le sens caché d'un type déclaré ou d'un ensemble d'entités d'un cluster consiste à l'expliquer à partir de ses instances. La capture du sens d'un type ne consiste pas à trouver un ensemble de synonymes au nom du type. En effet, un synonyme n'apporte pas d'information nouvelle, mais une autre manière de formuler les noms. L'annotation consiste plutôt à rechercher dans d'autres bases de connaissances des entités qui ressemblent aux entités du clusters et qui sont potentiellement décrites différemment. Ce processus d'annotation peut faire émerger des connaissances cachées, par exemple, le processus d'annotation d'un

groupe de *politiciens* a permis d'extraire comme annotations des *politiciens* avec un degré de pertinence à 1, des *auteurs* avec un degré de pertinence à 0.8, des *présidents* avec un degré de pertinence à 0.3, des *ministres* avec un degré de pertinence à 0.8, etc. Une information intéressante cachée qu'on peut déduire de cet exemple, est que la plus part des *politiciens* sont des *auteurs*.

Une source de données peut contenir certaines informations relatives au schéma, telles que des déclarations de types d'entités ou des hiérarchies de types. À partir de cette première couche de métadonnées décrivant la source de données, des bases de connaissances externes peuvent être utilisées pour extraire plus de métadonnées exprimant sa sémantique. Par exemple, en considérant les types définis dans une source de données et en utilisant une ontologie de domaine, on peut trouver les types génériques et spécifiques, les sous-propriétés, etc. Le processus d'extraction de cette connaissance supplémentaire est fondé sur les types définis dans le jeu de données, en cherchant des super-types ou des sous-types dans une base de connaissances. Une autre façon de fournir des connaissances supplémentaires sur le jeu de données est de considérer les instances des types. Dans ce cas, le sens d'un type est extrait en fonction de son contenu réel, et non sur la base de son nom.

L'annotation renseigne sur le sens des types, ce qui pourrait avoir plusieurs utilités. Les annotations découvertes peuvent être exploitées pour l'alignement de différents jeux de données. En effet, l'annotation de types peut aider à surmonter l'hétérogénéité entre les différents ensembles de données, cela en identifiant des types ayant la même sémantique en comparant leurs ensembles respectifs d'annotations. De la même manière, les annotations découvertes peuvent également être exploitées à des fins d'interconnexion entre les jeux de données.

Plusieurs travaux sur l'annotation de clusters ont été présentées dans l'état de l'art. Les approches qui concernent les données liées [2, 31, 26] ne fournissent pas de méthodes dédiées à l'annotation des entités mais plutôt une manière de trouver le label d'un groupe découvert par une approche d'extraction de schéma. Dans [31], les arcs entrants les plus fréquents pour les données OEM [25] sont considérés, ce qui équivaut à considérer les labels les plus courants pour les déclarations *rdf:type* ou *dcterm:subject* dans une source de données RDF, comme proposé dans [26, 2]. Cependant, ces déclarations ne sont pas toujours fournies. Les annotations produites par nos algorithmes peuvent être utilisées comme des noms candidats pour ces clusters. Les approches proposées dans le contexte des tables Web [48, 49, 51, 52], peuvent être adaptées aux données RDF, mais, contrairement aux données RDF, les données tabulaires sont structurées. L'idée principale des approches d'annotation de documents [54, 55, 56, 57, 58] est d'annoter un groupe de documents similaires avec les mots les plus fréquents contenus dans le texte de ces documents. Cependant, les annotations qui décrivent le mieux les données RDF ne sont pas nécessairement présentes dans le jeu de données, ni dans les propriétés, ni dans les valeurs de celles-ci.

Dans ce chapitre, nous abordons le problème de l'annotation des types d'une source de données RDF. Nous considérons pour cela les instances de ces types et nous utilisons des bases de connaissances disponibles sur le Web, qui fournissent une quantité d'informations sans précédent. Le but de notre travail est de fournir pour chaque type un ensemble d'annotations capturant la sémantique de ses instances et permettant de comprendre son contenu.

Les contributions présentées dans ce chapitre sont : (i) un ensemble d'algorithmes d'annotation fournissant pour chaque type contenu dans la source de données, les annotations sémantiques qui reflètent le mieux leur signification, et (ii) une approche qui exploite ces annotations pour découvrir la hiérarchie de types d'une source de données.

Ce chapitre est organisé comme suit. La formalisation du problème est présentée dans la section 5.2. Dans la section 5.3, nous présentons les bases de connaissances disponibles sur le Web qui ont été utilisées pour l'annotation. Nous décrivons ensuite les différents algorithmes d'annotation dans la section 5.4. La génération d'une hiérarchie de types à l'aide des annotations sémantiques du type, est décrite dans la section 5.5. Nos résultats d'évaluation sont présentés dans la section 5.6. Nous concluons ce chapitre en section 6.10.

5.2 Formalisation de la problématique

Rappelons qu'une source de données RDF est associée aux ensembles R, B, P et L représentant des ressources, des nœuds blancs (ressources anonymes), des propriétés et des littéraux respectivement. Une source de données décrite en RDF(S)/OWL est définie comme un ensemble de triplets $D \subseteq (R \cup B) \times P \times (R \cup B \cup L)$. Graphiquement, un jeu de données est représenté par un graphe orienté et étiqueté G, où chaque nœud est une ressource, un nœud blanc ou un littéral. Chaque arc dans le graphe, reliant un nœud e_1 à un autre nœud e_2 étiqueté par la propriété p représente le triple (e_1, p, e_2) du jeu de données D. Dans ce graphe, nous définissons une entité comme un nœud correspondant soit à une ressource qui peut être n'importe quel nœud en dehors de ceux correspondant aux littéraux.

Considérons le schéma S composé des ensembles T et I représentant respectivement les types et les liens entre eux. Chaque type $t_i \in T$ représente un groupe d'entités de même type dans le jeu de données. Un lien d'un type t_i à un autre type t_j représente une propriété pour laquelle le domaine est t_i et le co-domaine est t_j.

Ce schéma est déclaré dans le jeu de données lui-même en utilisant certaines propriétés spécifiques telles que *rdfs:domain*, *rdfs:range* et *rdf:type*. Notre but est d'annoter les types dans le jeu de données et de trouver les termes qui les expliquent le mieux en capturant la sémantique de leurs entités. L'annotation

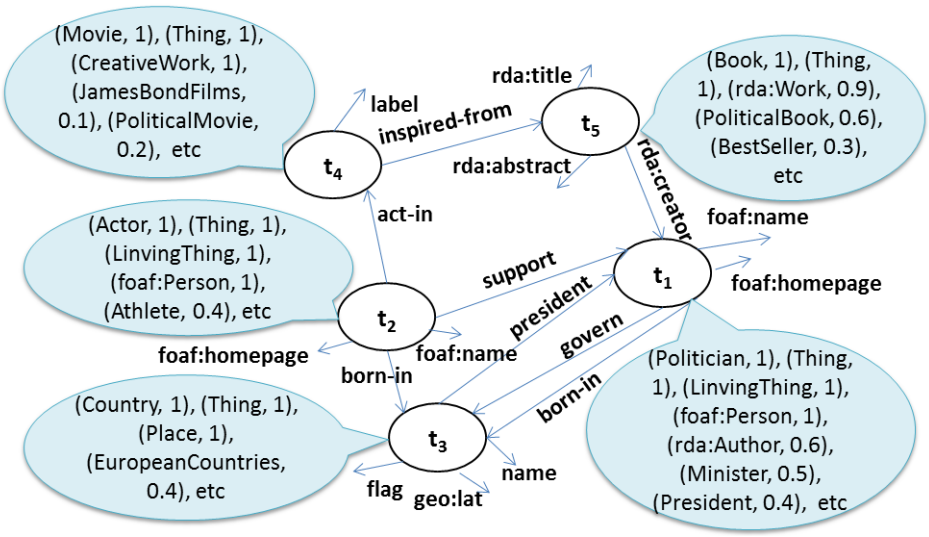


FIGURE 5.1 – Annotation de types d'un jeu de données RDF.

de type peut aider à surmonter l'hétérogénéité entre le schéma de différents ensembles de données, et cela en identifiant des types ayant la même sémantique.

Notre problème peut être énoncé comme suit. Considérons un schéma S décrivant un ensemble de types $T = \{t_1, ..., t_n\}$, où chaque type t_i correspond à un ensemble d'entités. Notre objectif est de trouver pour chaque type t_i un ensemble d'annotations $A^{t_i} = \{(a_1, w_1), ..., (a_k, w_k)\}$ qui décrit au mieux les instances du type. Chaque annotation a_j est associée à un poids w_j qui reflète la pertinence de l'annotation pour le type considéré. Les annotations sont extraites de bases de connaissances externes.

Une base de connaissances contient des informations liées au schéma. Ces informations peuvent être sous la forme de jeux de données liées ayant des déclarations sur le type des entités, comme dans le *Linked Open Data Cloud (LOD Cloud)* [20] qui est disponible sur le Web. Ces informations sur le schéma peuvent être également sous forme de vocabulaires, tels que FOAF (Friend Of A Friend) pour décrire des personnes et leurs liens les uns avec les autres, le vocabulaire RDA (Resource Description and Access) pour les données bibliographiques, le vocabulaire Basic Geo (WGS84 lat/long) pour les données spatiales, etc. En effet, un vocabulaire contient des informations sur le schéma comme les types et les liens entre eux.

La figure 5.1 montre un exemple de schéma décrivant une source de données RDF. Chaque type t_i représente un ensemble d'entités. Chaque bulle contient des annotations pour un type donné. Ces annotations permettent de comprendre la signification des types. Plus le poids de l'annotation est élevé, plus il est pertinent pour le type. En outre, les annotations peuvent fournir plus de connaissances sur les données que ce qui est spécifié dans le jeu de données. Par exemple, le type t_1

qui est déclaré comme *Politician* dans le jeu de données est également annoté par le label *Author* avec un poids élevé, montrant que la plupart des politiciens sont aussi des auteurs dans ce jeu de données. Les annotations montrent aussi que les entités de type *Politician* sont des présidents et des ministres, ce qui permet de mieux comprendre leur sens.

Les déclarations de type peuvent être incomplètes ou manquantes dans une source de données. Elles peuvent être découvertes en regroupant les entités selon la similarité de leur structure en utilisant des techniques de clustering, comme présenté dans le chapitre précédent. Les types découverts sont des clusters d'entités. L'annotation peut également être appliquée dans ce contexte pour extraire les étiquettes les plus représentatives pour les clusters, qui pourraient être utilisées comme noms des types candidats.

5.3 Bases de connaissances utilisées

Afin d'expliquer le sens d'un groupe d'entités (cluster/classe/type) nous proposons de l'annoter, en extrayant à partir de bases de connaissances externes, un ensemble de termes qui décrivent les entités du groupe.

Une façon de retrouver le sens d'une entité est de retrouver une déclaration pertinente de type pour cette entité ou pour une entité similaire. En effet, les déclarations sur le type d'une entité permettent de l'expliquer en partie. Celles-ci sont fournies dans certaines bases de connaissances disponibles sur le Web. Ces bases de connaissances peuvent se présenter sous la forme de jeux de données liées ayant des déclarations sur le type des entités, comme dans le *Linked Open Data Cloud (LOD Cloud)* [20] ; ou sous forme de vocabulaires standards, tel que FOAF (Friend Of A Friend), RDA (Resource Description and Access), etc. En effet, un vocabulaire contient des informations sur le schéma comme les types et les liens entre eux. Un ensemble de vocabulaires sont fournis sur le Web dans le *Linked Open Vocabulary* (LOV) [19, 13]. Nous considérons à la fois la base de connaissances et le vocabulaire comme un graphe de données RDF. Nous proposons d'extraire des annotations pour les types à partir de ces deux bases de connaissances disponibles sur le Web, comme décrit dans les sections 5.3.1 et 5.3.2.

5.3.1 Linked Open Data Cloud

Nous avons utilisé le *Linked Open Data Cloud* (*LOD Cloud*) comme base de connaissances, via un point d'accès SPARQL [20], car il fournit des données couvrant un large éventail de sujets différents. La base de connaissances peut être plus spécifique, par exemple *geonames*, si nous connaissons le domaine du jeu de données.

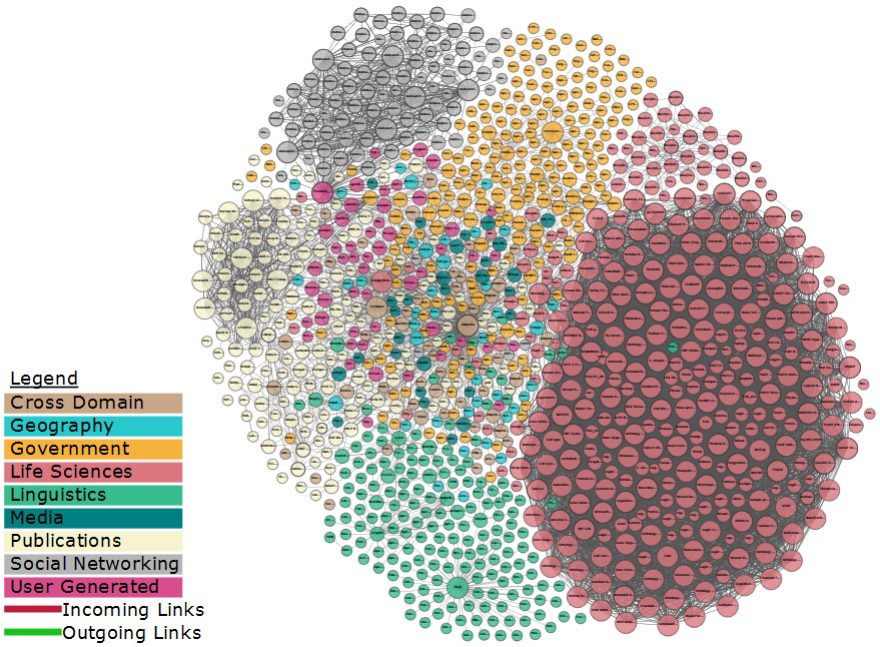


FIGURE 5.2 – Diagramme du *Linked Open Data Cloud* 2017 [12].

La figure 5.2 montre l'état du Linked Data Cloud en 2017. Celui-ci est constitué des jeux de données qui ont été publiés dans le format de données liées. Le diagramme est fondé sur les métadonnées recueillies et organisées par les contributeurs du DataHub. Chaque cercle représente une source de données dans le DataHub. La taille des cercles correspond au nombre de liens entre les jeux de données. Un lien est un triple RDF où les URI sujets et objets sont dans des espaces de noms de différents jeux de données.

5.3.2 Linked Open Vocabulary

Afin d'exploiter les vocabulaires utilisés dans le jeu de données, nous nous référons aux vocabulaires fournis par le *Linked Open Vocabularies (LOV)* [19, 13] à travers un point d'accès SPARQL [84]. Comme le montre la figure 5.3, celui-ci, regroupe 520 vocabulaires publiés par des organismes et des individus couvrant un large éventail de domaines.

LOV a été initié en 2011, dans le cadre d'un projet de recherche français, DataLift [85]. Son principal objectif était d'aider les éditeurs et les utilisateurs de données et de vocabulaires liés à évaluer ce qui était disponible pour leurs besoins, à le réutiliser autant que possible et à insérer leur propre production de vocabulaire de façon transparente dans l'écosystème.

Un vocabulaire dans LOV rassemble les définitions d'un ensemble de types et de propriétés, appelées simplement les termes du vocabulaire. Ces termes sont

utiles pour décrire les types spécifiques des entités dans un domaine donné, ou de manière générale pour un usage spécifique. Les vocabulaires sont également des données. En effet, grâce à la nature même du Web et au langage RDF, les vocabulaires peuvent s'exprimer comme des données du Web sémantique. Les termes de vocabulaire, sont identifiés par des URI publiques et peuvent être décrits en RDF.

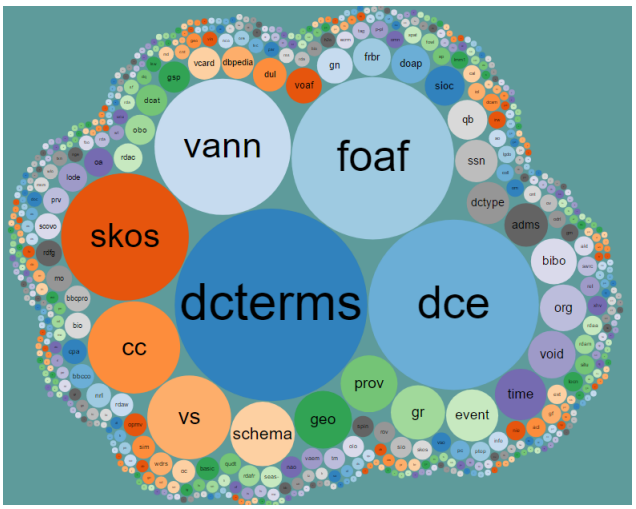


FIGURE 5.3 – Diagramme du *Linked Open Vocabulary* 2017 [13].

Plusieurs applications utilisent *LOV*, parmi elles :

- Datalift [85] qui est une plate-forme dédiée à la transformation des données. Dans Datalift, les données d'entrée sont des données brutes issues de multiples formats hétérogènes (bases de données, CSV, XML, RDF, RDFa, GML, Shapefile, etc). Les données de sortie produites sont des "données liées" qui utilisent les vocabulaires du *LOV*. La plate-forme Datalift est activement impliquée dans la mutation du Web vers les données liées.
- Online Ontology Set Picker (OOSP) [86] permet de sélectionner, à partir de référentiels majeurs, des ontologies qui satisfont un ensemble de paramètres définis par l'utilisateur. Son objectif principal est de permettre aux concepteurs d'outils ontologiques de construire rapidement leurs ontologies en utilisant les vocabulaires du *LOV*.
- OntoWiki [87] facilite la présentation visuelle d'une base de connaissances en tant que carte d'information, avec différentes vues sur les données. Il permet la création intuitive de contenu sémantique, avec un mode d'édition en ligne de contenu RDF utilisant les vocabulaires du *LOV*.

5.4 Les algorithmes d'annotation de types

Afin d'annoter les types d'une source de données RDF, nous proposons trois algorithmes : l'annotation utilisant le nom, l'annotation utilisant les propriétés et l'annotation utilisant le vocabulaire.

L'annotation utilisant le nom extrait le type des entités de la base de connaissances ayant le même nom que les entités du type. L'annotation utilisant les propriétés recherche des entités dans la base de connaissances possédant une propriété similaire à l'une des propriétés du type. Enfin, l'algorithme d'annotation utilisant le vocabulaire renvoie les annotations pour les types décrits par un vocabulaire standard tel que le type t_2, dans la figure 5.1, qui est décrit par la propriété *foaf:homepage* du vocabulaire standard FOAF. L'algorithme d'annotation utilisant le vocabulaire exploite les propriétés standards pour faire correspondre les types du jeu de données aux types déclarés dans les vocabulaires. Nous proposons également une approche hybride qui combine les ensembles d'annotations retournées par chaque algorithme.

Cette section est organisée comme suit. Nous proposons dans la section 5.4.1, une approche d'annotation utilisant le nom des instances d'un groupe. Dans la section 5.4.2, nous présentons notre approche d'annotation utilisant les propriétés des instances d'un groupe. Dans la section 5.4.3, nous présentons notre approche d'annotation utilisant les propriétés issues des vocabulaires. Nous présentons également, dans la section 5.4.4, une approche hybride qui combine les ensembles d'annotations retournées par chaque algorithme.

5.4.1 Annotation utilisant le nom

L'annotation utilisant le nom s'appuie sur une base de connaissances et considère le nom des entités du type pour expliquer son contenu. Nous avons utilisé le *LOD Cloud* comme base de connaissances, mais toute autre base de connaissances peut être utilisée. Étant donnée une entité e dans le jeu de données, nous cherchons dans la base de connaissances une entité e' ayant le même nom que e. Nous extrayons les types de e' comme annotations possibles pour caractériser e. Le nom d'une entité représente la valeur de la propriété *name* ou de toute autre propriété représentant les mêmes informations, comme la propriété *label* pour le type t_4 et la propriété *title* pour le type t_5 de la figure 5.1. Un exemple du processus d'annotation utilisant le nom en considérant le type t_4 est présenté dans la figure 5.4. Pour l'entité e_1, les annotations *Movie* et *Creative Work* ont été extraites en recherchant les types des entités ayant la même valeur pour la propriété *label* dans le *LOD Cloud*.

Chaque entité peut avoir plusieurs types. Par exemple, les types trouvés pour l'entité *Barack Obama* sont *Président*, *Person*, *Politician*, etc., et tous ces types sont des annotations pertinentes pour l'entité. Pour trouver les annotations perti-

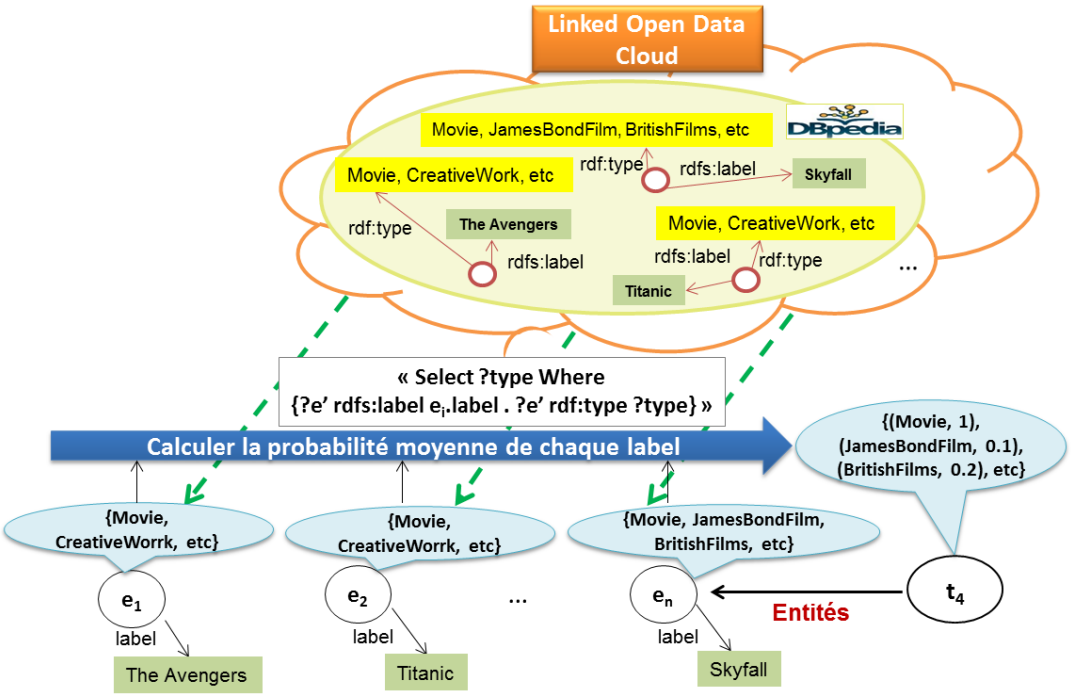


FIGURE 5.4 – Annotation utilisant le nom.

nentes pour un cluster d'entités, nous considérons les types les plus fréquemment renvoyés pour toutes les entités de ce cluster.

Un ensemble d'annotations est renvoyé pour chaque entité du type et le résultat final de notre algorithme est l'ensemble des $top-k$ annotations les plus fréquentes parmi celles retournées pour les entités du type. Nous assignons une probabilité pour chaque annotation a calculée comme la proportion d'entités du type pour laquelle a a été retournée. Pour chaque annotation, un poids est calculé pour refléter la fréquence de l'annotation parmi les instances du type. Par exemple, dans la figure 5.4, le poids 1 pour l'annotation *Movie* indique que cette annotation a été extraite pour chaque instance du type t_4. L'ensemble des annotations pour une entité et pour un type est défini ci-dessous.

Définition (Ensemble d'annotations déduites par le nom pour une entité). Pour une entité e_i, où $e_i.name$ représente la valeur de la propriété $name$; l'ensemble d'annotations A_i extrait à partir d'une base de connaissances externe KB est défini comme suit :

– $a \in A_i$, si $\exists$ $((e'_i,$ name, $e_i.name)$ et $(e'_i,$ rdf:type, $a)) \in KB$

Définition (Ensemble d'annotations déduites par le nom pour un type). Considérons un ensemble d'entités d'un type $t = \{e_1, ..., e_n\}$ et leurs ensembles respectifs d'annotations $A = \{A_1, ..., A_n\}$. Nous définissons l'ensemble

d'annotations $A^t = \{(a_1, w_1), ..., (a_k, w_k)\}$ pour le type t comme les $top-k$ paires (a_i, w_i) ordonnées selon leur poids w_i. Nous définissons $(a_i, w_i) \in A^t$ comme suit :

- $a_i \in \{A_1 \cup A_2 \cup ...A_n\}$, i.e une annotation extraite pour une entité $e_j \in t$,
- $w_i = \sum_{\forall e_j \in t \wedge a_i \in A_j} 1 \ / \ |t|$, i.e la proportion d'entités dans t pour lesquelles a_i est extraite.

Comme décrit dans l'**algorithme 13**, pour chaque type du jeu de données, nous interrogeons la base de connaissances *LOD Cloud*, via un point d'accès SPARQL, pour récupérer pour chaque entité les types des entités dans le *LOD Cloud* ayant la même valeur pour la propriété *name*. Ensuite, nous assignons une probabilité pour chacun des $top-k$ types renvoyés.

Algorithm 13: Annotation utilisant le nom

Input : La base de connaissances KB, un type t, k
Output: L'ensemble d'annotations A^t

for $\forall e_i \in t$ **do**
  Rechercher A_i à partir de KB :
  A_i = *SELECT distinct ?type WHERE { ?e_i' name e_i.name . ?e_i' rdf:type ?type}*;
  $A = A \cup A_i$;
end
for $\forall$ *différents* $a_i \in A$ **do**
  w_i = (nombre de $a_i \in A$)/$|t|$;
  $A^t = A^t \cup (a_i, w_i)$;
end
Garder les k annotations avec les poids les plus élevés dans A^t;

Certaines des entités d'un type peuvent ne pas être décrites par une propriété *name*, ou une propriété représentant les mêmes informations, et même si elle est disponible, elle peut être inconnue dans la base de connaissances. Dans une telle situation, l'annotation utilisant le nom échouera dans la recherche des annotations pertinentes. Pour remédier à ce problème, nous proposons, dans la section suivante, un autre algorithme d'annotation utilisant les ensembles de propriétés des entités.

5.4.2 Annotation utilisant les propriétés

De même que l'annotation utilisant le nom, l'annotation utilisant les propriétés repose sur une base de connaissances pour trouver les annotations pertinentes pour un type. Nous avons utilisé le *LOD Cloud* comme base de connaissances, mais tout autre base de connaissances peut être utilisée. Pour un type donné t, nous interrogeons la base de connaissances afin de trouver des entités ayant une

propriété similaire à l'une des propriétés des instances de t. Par exemple, dans la figure 5.1, le type t_3 a la propriété entrante *born-in* qui donne l'intuition que le type est *Place*. De plus, ce type est décrit par la propriété sortante *flag* qui indique que l'annotation est *Country*.

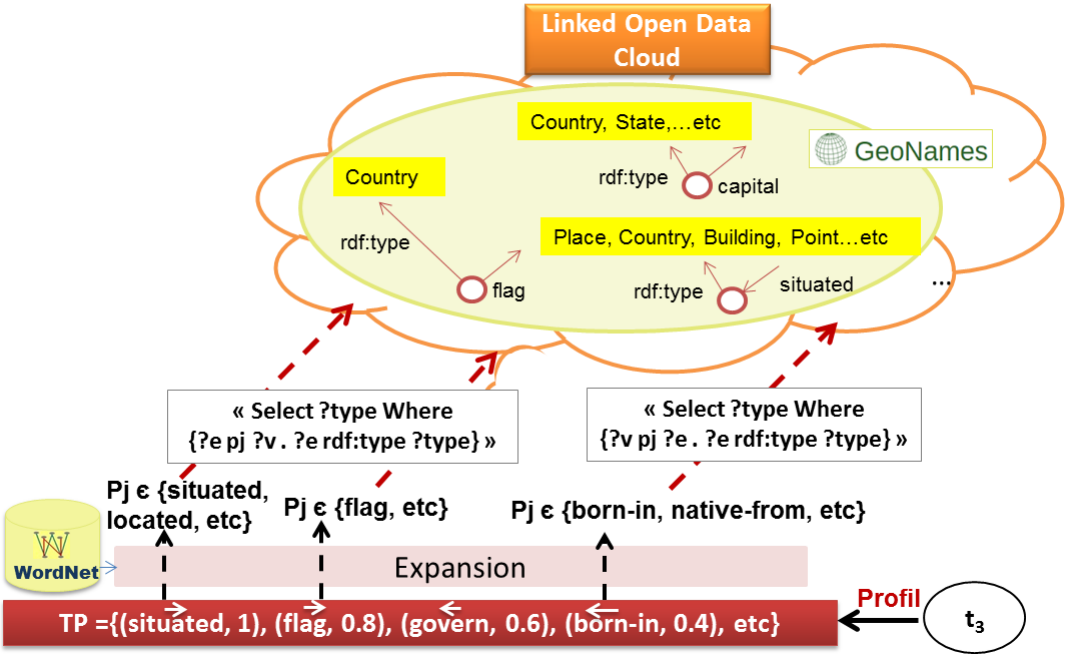


FIGURE 5.5 – Annotation utilisant les propriétés.

Les entités RDF de même type n'ont pas exactement les mêmes ensembles de propriétés. Nous considérons que plus une propriété est utilisée pour décrire les entités d'un type, plus elle reflète la sémantique de ce type. Nous utilisons la notion de profil de type définie dans le chapitre 3 : $TP = \{(p_1, \alpha_1), ..., (p_n, \alpha_n)\}$. Ce profil de type est constitué d'un ensemble de propriétés avec leurs probabilités. Chaque p_i représente une propriété et chaque α_i représente la probabilité pour une entité de t d'avoir la propriété p_i. La probabilité α_i associée à une propriété p_i dans le profil du type t est évaluée comme le nombre d'entités du type t pour laquelle p_i est définie sur le nombre total d'entités dans t.

La figure 5.5 illustre l'annotation utilisant les propriétés pour le type t_3 de la figure 5.1. Pour obtenir les annotations d'un type, pour chaque propriété dans le profil de type, nous recherchons dans le *LOD Cloud* des entités ayant la même propriété et nous extrayons leurs types. Les propriétés ayant la même signification peuvent avoir des terminologies différentes dans le jeu de données et la base de connaissances. Pour surmonter cette hétérogénéité, nous générons un ensemble de synonymes pour chaque propriété en utilisant *WordNet* [88]. Ensuite, une recherche est effectuée dans la base de connaissances pour chaque synonyme jusqu'à ce que la propriété correspondante soit retrouvée dans la base de connaissances.

Notons que le problème de définir les correspondances entre les propriétés du jeu de données et les propriétés de la base de connaissances n'est pas adressé dans ce document.

Algorithm 14: Annotation utilisant les propriétés

Input : La base de connaissances KB, un type t, k
Output: L'ensemble d'annotations A^t

Construire le profil de type $TP = \{(p_1, \alpha_1), ..., (p_n, \alpha_n)\}$;
for $\forall$ $(p_j, \alpha_j) \in TP$ **do**
  $synonymSet = p_j \cup extractSynonym(p_j)$ à partir de **WordNet**;
  for $\forall$ $p'_j \in synonymSet$ **do**
    if p_j *est une propriété sortante* **then**
      A_j = *SELECT ?type WHERE { ?e* p'_j *?v . ?e rdf:type ?type}* (à partir de KB);
    else
      A_j = *SELECT ?type WHERE { ?e* p'_j *?v . ?v rdf:type ?type}* (à partir de KB);
    end
    $A = A \cup A_j$;
  end
end
for $\forall$ *différents* $a_i \in \{A_1 \cup ... A_n\}$ **do**
  $w_i = 0$;
  for $\forall A_j \in A$ **do**
    if $a_i \in A_j$ **then**
      $w_i = w_i + \alpha_j$;
    end
  end
  $w_i = w_i \ / \ |TP|$;
  $A^t = A^t \cup (a_i, w_i)$;
end
Garder les k annotations avec les poids les plus élevés dans A^t;

Une propriété peut être entrante ou sortante, le domaine et le co-domaine d'une propriété sont donc importants lors de l'interrogation de la base de connaissances. Les annotations retournées pour une propriété sont pondérées par la probabilité de la propriété dans le profil. Ceci est dû au fait que plus la probabilité d'une propriété est élevée, mieux cette dernière reflète la sémantique du type. L'ensemble des annotations pour une propriété et l'ensemble d'annotations pour un type sont définis ci-après.

Définition (Ensemble d'annotations pour une propriété). L'ensemble A_i des annotations extraites pour une propriété p_i à partir d'une base de connaissances externe KB à l'aide de l'annotation utilisant les propriétés est défini selon la direction de la propriété, comme suit :

- $\overrightarrow{p_i}$: Si $\exists$ ((e, p_i, v) et (e, rdf:type, a)) $\in KB$ alors ($a \in A_i$)
- $\overleftarrow{p_i}$: Si $\exists$ ((e, p_i, v) et (v, rdf:type, a)) $\in KB$ alors ($a \in A_i$)

Définition (Ensemble d'annotations utilisant des propriétés pour un type). Étant donné le profil $TP = \{(p_1, \alpha_1), ..., (p_n, \alpha_n)\}$ d'un type t et les ensembles respectifs d'annotations A_i pour chaque propriété p_i dans TP, tels que $A = \{A_1, ..., A_n\}$, nous définissons un ensemble d'annotations pour t annoté $A^t = \{(a_1, w_1), ..., (a_k, w_k)\}$, comme les $top - k$ paires (a_i, w_i) ordonnées selon leur poids w_i de l'annotation a_i. Nous définissons $(a_i, w_i) \in A^t$ comme suit :

- $a_i \in \{A_1 \cup A_2 \cup ...A_n\}$, i.e une annotation extraite pour une propriété décrivant TP ;
- $w_i = \sum_{\forall (p_j, \alpha_j) \in TP \wedge a_i \in A_j} \alpha_j \ / \ |TP|$, i.e la somme des probabilités des propriétés pour lesquelles a_i est extraite sur le nombre de propriétés dans TP.

Certaines propriétés peuvent renvoyer la même annotation, dans ce cas nous calculons la somme des probabilités des propriétés pour lesquelles cette annotation a été générée. Cette somme est divisée par le nombre total de propriétés dans le profil pour normaliser le poids, et les k annotations correspondants aux poids les plus élevés sont retournées, comme décrit dans l'**algorithme 14**.

Lorsque le jeu de données utilise des vocabulaires standards, cela réduit considérablement les problèmes d'hétérogénéité. En effet, un vocabulaire est utilisé pour définir des types et des propriétés standards, qui peuvent être utilisés pour dériver des annotations pertinentes. Lorsque des données sont décrites en RDF, il est préférable d'utiliser autant que possible des types et des propriétés déjà définis dans des vocabulaires existants pour assurer l'interopérabilité avec une source de données externe.

5.4.3 Annotation utilisant les vocabulaires

L'annotation utilisant le vocabulaire repose sur les vocabulaires standards utilisés pour décrire le jeu de données. Un vocabulaire RDF est un ensemble prédéfini de prédicats qui peuvent être utilisés pour décrire des entités. Un vocabulaire est défini en créant une ontologie qui contient tous les types et propriétés possibles avec leur domaine et co-domaine. Il est généralement conçu pour un domaine spécifique, tel que FOAF (Friend Of A Friend) pour décrire des personnes et leurs liens les uns avec les autres, le vocabulaire RDA (Resource Description and Access) pour les données bibliographiques, le vocabulaire Basic Geo (WGS84

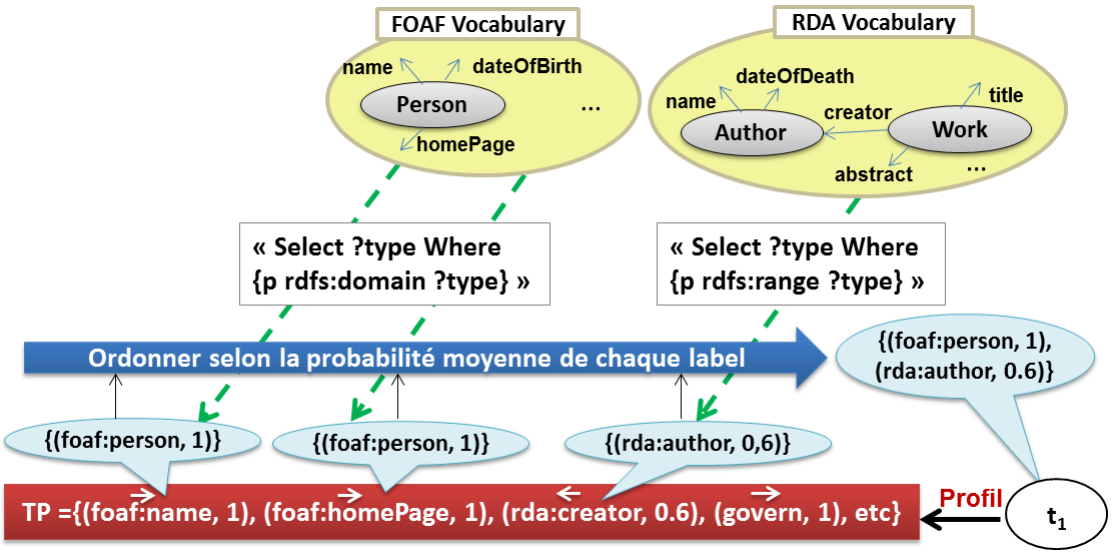


FIGURE 5.6 – Annotation d'un type en utilisant des vocabulaires standards.

lat/long) pour les données spatiales, etc. D'autres vocabulaires sont plus généraux tels que SKOS (Simple Knowledge Organization System) pour décrire des référentiels de types thésaurus ; et VOID (Vocabulary Of Interlinked Datasets) qui fournit des métadonnées sur les jeux de données RDF. Ils ne sont pas conçus pour un domaine spécifique et les types qu'ils contiennent sont suffisamment généraux pour être appliqués à n'importe quelle ressource. Nous ne considérons pas les vocabulaires qui sont généraux dans notre approche, nous nous concentrons plutôt sur des vocabulaires de domaines spécifiques, car ils fournissent des annotations plus précises. Nous avons utilisé les vocabulaires fournis dans la source *Linked Open Vocabulary* (LOV) [19, 13] car elle fournit des vocabulaires couvrant un large éventail de sujets publiés par des organismes standards et des individus.

La figure 5.6 montre l'idée générale de notre processus d'annotation utilisant le vocabulaire pour le type t_1 de la figure 5.1. Le type t_1 est décrit par la propriété $\overrightarrow{foaf : name}$ et la propriété $\overrightarrow{foaf : homePage}$ du vocabulaire FOAF. Il est également décrit par la propriété $\overleftarrow{rda : creator}$ du vocabulaire RDA. On peut voir certains types et propriétés avec leurs domaines et leurs co-domaines dans FOAF et RDA. Nous recherchons dans les vocabulaires le domaine et le co-domaine des propriétés comme des annotations possibles pour un type. Par exemple, le domaine de $\overrightarrow{foaf : name}$ dans FOAF est *Person* qui représente une annotation possible pour le type t_1, et le co-domaine de la propriété $\overleftarrow{rda : creator}$ dans RDA est *Author* qui représente également une annotation possible pour le type. Chaque annotation possible pour un type est pondérée en fonction des probabilités des propriétés dans son profil, en considérant les propriétés pour lesquelles cette annotation a été extraite, de la même manière que pour l'annotation utilisant les propriétés (voir section 5.4.2).

Le vocabulaire est utilisé de la même façon que la base de connaissances dans l'**algorithme 14**. Cependant, la base de connaissances est interrogée pour trouver des entités, tandis que le LOV est interrogé pour récupérer le domaine et le co-domaine des propriétés. En effet, les vocabulaires ne contiennent pas d'entités et les bases de connaissances ne disposent pas nécessairement d'informations complètes sur le domaine et le co-domaine des propriétés. La requête correspondante sera formulée selon le sens de la propriété comme illustré dans la figure 5.6.

5.4.4 Annotation hybride

Les algorithmes d'annotation précédents capturent la sémantique d'un type de différentes manières : l'annotation utilisant le nom génère un ensemble d'annotations pondérées $A_N = \{(a_1, \gamma_1), .. ., (A_n, \gamma_n)\}$, en fonction du nom des entités du type ; l'approche d'annotation utilisant les propriétés produit un ensemble d'annotations pondérées $A_P = \{(a_1, \beta_1), ..., (a_p, \beta_p)\}$, en fonction des propriétés du profil de type ; enfin, l'approche d'annotation utilisant le vocabulaire produit un ensemble d'annotations pondérées $A_V = \{(a_1, \sigma_1), ..., (a_v, \sigma_v)\}$, en fonction des propriétés standards du profil de type.

Nous proposons de pondérer chaque ensemble d'annotations selon la pertinence de l'algorithme utilisé. Soit $TP = \{(p_1, \alpha_1), ..., (p_n, \alpha_n)\}$ le profil d'un type t à annoter à partir d'une base de connaissances KB ; le poids de chaque ensemble d'annotation o_N, o_P, o_V utilisant respectivement le nom, les propriétés et le vocabulaire, est calculé comme suit :

- $o_N = \sum_{\forall e_i \in c \wedge e_i.name \in KB} 1 \ / \ |t|$, i.e la proportion d'entités dans t dont le nom est retrouvé dans KB ;
- $o_P = \sum_{\forall (p_j, \alpha_j) \in TP \wedge p_j \in KB} 1 \ / \ |TP|$, i.e la proportion de propriétés dans TP, ou leurs équivalentes, qui ont été retrouvées dans KB ;
- $o_V = \sum_{\forall (p_v, \alpha_v) \in TP \wedge p_v \in LOV} 1 \ / \ |TP|$, i.e la proportion de propriétés standards dans TP.

Chaque approche donne un ensemble d'annotations qui pourraient être fusionnées pour produire un ensemble global et hybride. Soit l'ensemble d'annotations hybrides $A_H = \{(a_1, \chi_1), ..., (a_h, \chi_h)\}$ dans lequel la pertinence de chaque annotation est évaluée comme la moyenne pondérée de chaque ensemble d'annotations en fonction de la pertinence de l'algorithme d'annotation, comme décrit dans la formule 5.1.

$$\forall \lambda \in]0, 1], \forall (a_i, \lambda) \in (A_N \cup A_P \cup A_V) : \chi_i = \frac{o_N.\gamma_i + o_P.\beta_i + o_V.\sigma_i}{o_N + o_P + o_V} \qquad (5.1)$$

5.5 Utilisation des annotations pour la découverte de la hiérarchie des types

La hiérarchie des types dans une source de données contribue également à mieux comprendre et organiser le contenu. Nous proposons de découvrir cette hiérarchie et d'annoter les super-types en analysant les ensembles d'annotations obtenues à l'aide de l'approche d'annotation hybride d'un type présentée dans la section 5.4.4. Celle-ci combine les algorithmes d'annotation utilisant le nom, les propriétés et le vocabulaire qui sont décrits, respectivement, dans les sections 5.4.1, 5.4.2 et 5.4.3.

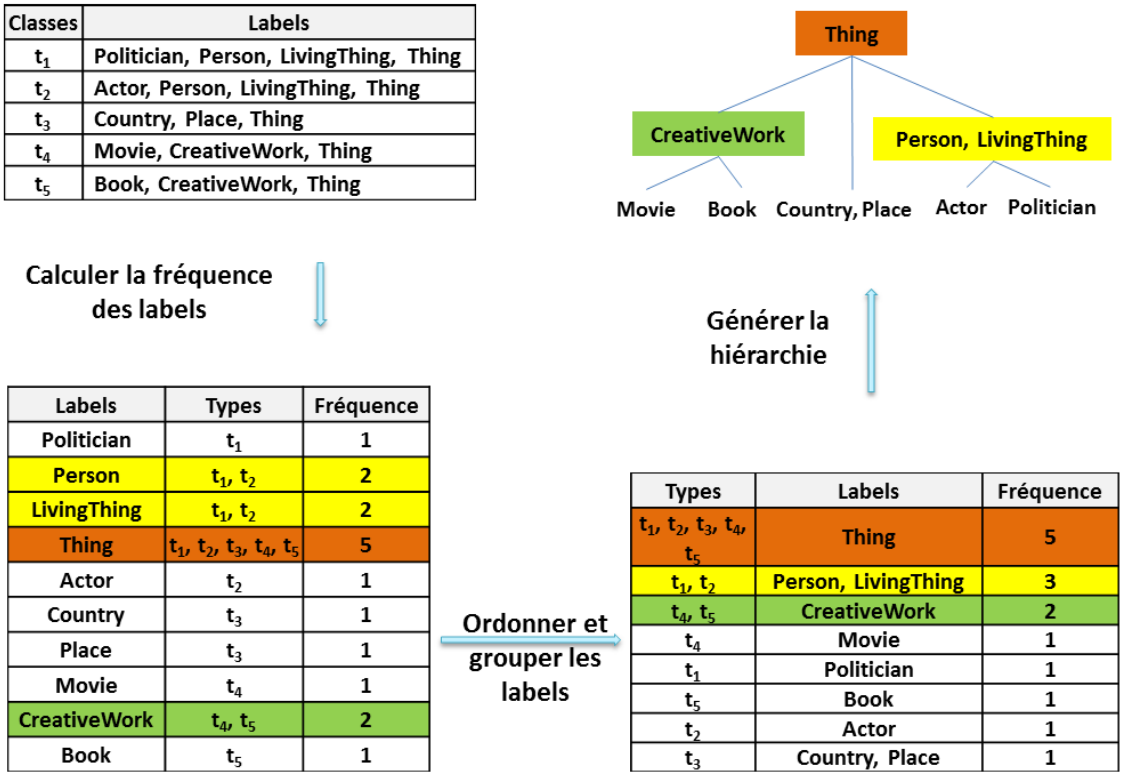


FIGURE 5.7 – Processus de découverte de la hiérarchie des types et de leurs annotations.

Nous considérons que des types ont un super-type commun si ces types partagent certaines de leurs annotations avec un poids égal à 1. Ce qui signifie que ces annotations sont pertinentes pour toute entité des types considérés. La figure 5.7 illustre le processus de découverte de la hiérarchie pour le jeu de données de la figure 5.1. Dans notre exemple, le type t_1 qui représente *Politician* et le type t_2 qui représente *Actor* ont tous les deux l'annotation *Person*. Ce qui indique que cette annotation décrit les deux types et qu'elle peut donc représenter leur type

générique.

Notre approche pour découvrir les super-types du jeu de données suit la même idée que la détection des items-sets fréquents dans l'algorithme *Apriori* [89], bien connu pour l'extraction des règles d'association. Cependant, notre objectif ici n'est pas d'extraire des règles d'association, mais de détecter les annotations qui apparaissent fréquemment ensembles. Avoir un ensemble fréquent d'annotations signifie que cet ensemble est partagé entre plusieurs types.

Algorithm 15: Découverte et annotation de la hiérarchie de types

Input : Les ensembles d'annotation pour tous les types $A^T = \{A^{t1} \cup A^{t2} \cup ...A^{tn}\}$

for $\forall\ (a_j, 1) \in A^T$ **do**
  $F = F \cup \{(a_j$, calculer la fréquence de chaque $(a_j, 1) \in A^T)\}$;
end
Ordonner F en fonction de la fréquence croissante de chaque a_j;
Regrouper les annotations ayant la même fréquence qui décrivent les mêmes types dans M;
for $\forall\ S \in M$ *selon l'ordre croissant de leur fréquence* **do**
  if S *a une fréquence* > 1 **then**
    Créer un super-type annoté par S;
    Créer un lien (*rdfs:subClassOf*) entre S et les types qu'il décrit;
  end
end

L'**algorithme 15** décrit notre approche pour découvrir et annoter les super-types dans une source de données RDF. Pour détecter les types qui partagent certaines de leurs annotations, nous calculons la fréquence de chaque annotation telle qu'elle est représentée dans la figure 5.7. Ensuite, nous ordonnons les annotations en fonction de leur fréquence et groupons celles qui décrivent les mêmes types. Certaines annotations ne décrivent qu'un seul type et par conséquent elles sont considérées comme le nom des types. Un super-type est créé entre les types décrits par un ensemble d'annotations de fréquence supérieure à 1. L'ordre des super-types dans la hiérarchie suit la fréquence de l'ensemble d'annotations. Plus la fréquence est élevée, plus l'annotation est élevée dans la hiérarchie des types.

Nous avons proposé dans le chapitre 3 une approche de découverte de liens hiérarchiques entre les types découverts. Elle consiste en une adaptation de l'algorithme de clustering hiérarchique pour qu'il puisse être appliqué sur les profils de types. L'approche proposée dans ce chapitre consiste à détecter les annotations partagées entre plusieurs types pour inférer leur super-type.

5.6 Évaluation

Cette section présente quelques résultats d'expérimentation pour notre approche. Nous avons évalué la qualité des annotations de type fournies par les différents algorithmes d'annotation proposés : l'algorithme d'annotation utilisant les noms, l'algorithme d'annotation utilisant les propriétés, l'algorithme d'annotation utilisant les vocabulaires et algorithme d'annotation hybride qui combine les résultats des trois premiers algorithmes. Nous avons extrait des annotations pour les types à partir du *Linked Open Data Cloud* (*LOD Cloud*) via un point d'accès SPARQL [20], car il fournit des données couvrant un large éventail de sujets différents. Nous avons utilisé le *Linked Open Vocabulary* (LOV) [19, 13], à travers un point d'accès SPARQL [84], comme base de connaissances pour les vocabulaires, car il fournit 520 vocabulaires publiés par des organismes standards et des individus couvrant un large éventail de domaines. Nous avons utilisé les annotations générées pour les types pour découvrir la hiérarchie de types. Pour effectuer nos tests, nous avons utilisé différents jeux de données réels décrits dans la section suivante.

5.6.1 Les jeux de données

Nous avons utilisé trois jeux de données réels : le jeu de données *Conference*[1] qui expose les données de plusieurs conférences et ateliers du Web sémantique, ayant comme types : *Presentation*, *TutorialEvent*, *InProceedings*, *Point*, *Organization*, *Person*, etc ; le jeu de données *BNF*[2] fourni par la Bibliothèque Nationale de France, contenant les types suivants : *Expression*, *Concept*, *Manifestation*, *Person*, *work* ; et nous avons sélectionné un ensemble d'entités des types suivants dans *DBpedia*[3] : *Politician*, *FootballPlayer*, *Museum*, *Film*, *Game* and *Stadium*. Ces types de *DBpedia* ont été sélectionnés sur la base de l'existence d'un lien hiérarchique entre elles. Ces ensembles de données font partie du *LOD Cloud*. Évidemment, ils sont ignorés du *LOD Cloud* lors du processus d'annotation afin que leurs propres déclarations de types ne soient pas prises en compte pour évaluer la qualité des annotations retournées.

5.6.2 Métriques et méthodologie expérimentale

Pour évaluer la qualité des résultats fournis par nos algorithmes, nous avons considéré chaque type déclaré dans le jeu de données et l'ensemble d'instances associé. Nous avons extrait les déclarations de noms existantes de ces types de nos ensembles de données et nous les avons considérées comme une référence. Ensuite,

1. Conference : data.semanticweb.org/dumps/conferences/dc-2010-complete.rdf
2. BNF : datahub.io/fr/dataset/data-bnf-fr
3. DBpedia : dbpedia.org

nous avons exécuté nos algorithmes sur les jeux de données sans les déclarations de nom des types.

Pour évaluer la précision entre les annotations de référence et les annotations proposées, nous établissons des correspondances dans une base de données lexicale. Pour l'expérimentation, nous avons utilisé Java WordNet Similarity Library [90] qui exploite WordNet [88]. Par exemple, dans nos expérimentations, le type existant *Politician* et l'annotation générée *Person* ont une similarité sémantique de $0,88$.

Considérons l'ensemble des annotations d'un type t_i, ordonnées par poids décroissants, en considérant $A_i = \{a_1, ..., a_m\}$ et n_i le nom de référence du type. Nous avons évalué la précision de l'annotation au rang k comme dans la formule 5.2.

$$Précision(A_i, n_i)@k = SimilaritéSémantique(a_k, n_i) \tag{5.2}$$

Nous considérons que la précision globale des annotations d'un ensemble de types $T = \{t_1, ..., t_n\}$, pour les annotations au rang k est la précision moyenne de chaque annotation au rang k pour chaque type t_i, comme dans la formule 5.3.

$$Précision@k = \frac{\sum_{i=1}^{|T|} Précision(A_i, n_i)@k}{|T|} \tag{5.3}$$

Nous avons également évalué la qualité de la hiérarchie des types découverte. La précision des super-types découverts est évaluée par la similarité sémantique entre l'annotation des super-types découverts et le nom des super-types déclarés dans le jeu de données.

5.6.3 Résultats

La précision des annotations découvertes pour les types est représentée dans la figure 5.8. L'approche utilisant le nom donne de meilleurs résultats que les approches utilisant les propriétés et le vocabulaire. Cela signifie que les annotations récupérées à l'aide du nom sont plus précises que celles récupérées à l'aide des propriétés. Pour les ensembles de données de *Conference* et *BNF*, l'ensemble des annotations est correctement classé. En effet, les premières annotations sont les plus exactes. Cependant, pour *DBpedia*, les annotations les plus précises sont généralement au troisième et quatrième rang pour les annotations utilisant le nom et les propriétés. Cela est dû au fait que les premières annotations sont trop générales, comme les annotations : *Thing*, *LivingThing*, *Person* au lieu de l'annotation *Politician* par exemple. *DBpedia* a la précision la plus faible pour l'annotation utilisant le vocabulaire car le jeu de données utilise moins de vocabulaires standards pour décrire ses données que les ensembles de données *Conference* et *BNF*. Pour les données de *BNF* et *DBpedia*, l'approche utilisant les vocabulaires ne renvoie que trois annotations, car un vocabulaire a moins de types qu'une base

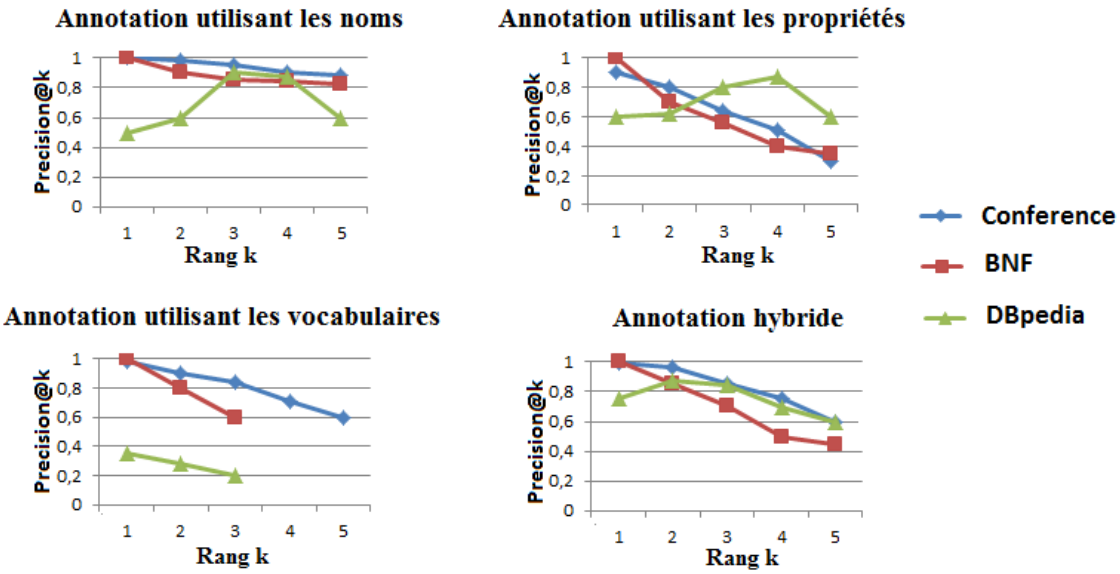


FIGURE 5.8 – Précision des annotations extraites pour les types.

de connaissances. L'approche hybride améliore le rang des annotations qui ont été renvoyées par les trois algorithmes d'annotation en augmentant leur poids. Elle donne une meilleure précision parce que les annotations communes entre les trois approches sont les plus précises.

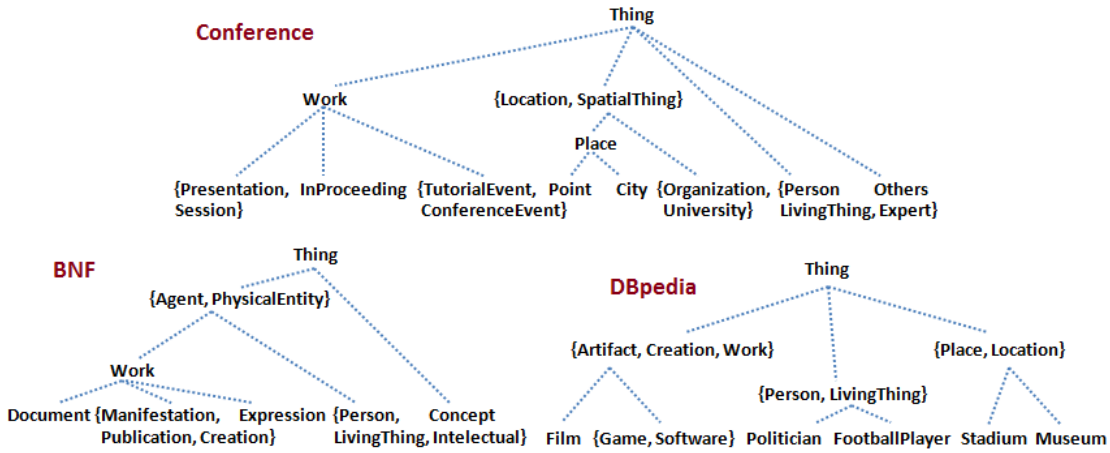


FIGURE 5.9 – Découverte de la hiérarchie des types.

La figure 5.9 montre la hiérarchie découverte et la figure 5.10 montre la précision de l'annotation des super-types. La précision des annotations des super-types de *DBpedia* est bonne et elles sont conformes aux déclarations *rdfs:subClassOf* du jeu de données. Pour le jeu de données *Conference*, la précision est également bonne, mais notre algorithme n'a pas pu identifier le super-type de *HomePage* et

PublicationPage car leurs profils de type diffèrent par une seule propriété seulement : $\overleftarrow{url}$ et $\overleftarrow{homepage}$. La précision des annotations des super-types de *BNF* est indéfinie car aucun lien hiérarchique n'est spécifié dans le jeu de données.

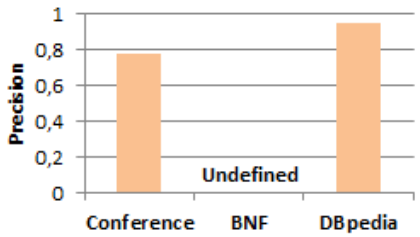


FIGURE 5.10 – Précision de l'annotation des super-types.

5.7 Conclusion

Nous avons proposé une approche pour annoter les types d'une source de données RDF. Cette approche fournit un ensemble d'annotations qui capturent la sémantique de ces types. Nous avons proposé quatre algorithmes d'annotation reposant sur des sources externes de connaissances : l'annotation utilisant le nom, l'annotation utilisant les propriétés, l'annotation utilisant le vocabulaire et l'annotation hybride qui combine les résultats des trois premiers algorithmes d'annotation. Chaque algorithme utilise les instances d'un type pour extraire ses annotations candidates. Nous avons également proposé une approche pour découvrir la hiérarchie de types dans une source de données à l'aide des annotations extraites pour les types.

Notre approche d'annotation peut être considérée comme un processus de génération de métadonnées sur les types d'une source de données RDF. Lorsque les déclarations de type sont manquantes, les techniques de regroupement peuvent être utilisées pour générer des clusters d'entités représentant les types candidats comme décrit dans le chapitre précédent et dans les articles [1, 31, 26]. Dans cette situation, les algorithmes d'annotation proposés peuvent être utilisés pour générer des noms de type candidats pour les clusters résultants.

Notre approche pour l'annotation de types pourrait également compléter les approches existantes pour la découverte de schéma [1, 31, 26]. La plupart de ces approches ne fournissent pas une méthode spécifique pour produire le nom pour les types générés. Dans [31], les arcs entrants les plus fréquents pour les données OEM [25] sont considérés, ce qui équivaut à considérer les labels les plus courants pour les déclarations *rdf:type* dans une source de données RDF, tel que proposé dans [26]. Cependant, ces déclarations ne sont pas toujours fournies. Les annotations produites par nos algorithmes peuvent être utilisées comme noms candidats pour ces clusters.

Les propriétés ayant la même signification peuvent avoir des terminologies différentes dans le jeu de données et la base de connaissances. Pour surmonter cette hétérogénéité, nous avons proposé de générer un ensemble de synonymes pour chaque propriété en utilisant WordNet [88]. Ensuite, nous effectuons une recherche dans la base de connaissances pour chaque synonyme jusqu'à ce que la propriété correspondante dans la base de connaissances soit retrouvée. Cependant, la génération de synonymes ne garantit pas la résolution du problème de correspondance entre la propriété du jeu de données et la propriété de la base de connaissances.

Dans ce chapitre, nous avons considéré des données RDF, mais notre approche d'annotation pourrait être adaptée à d'autres types de données telles que les données relationnelles, les tables Web ou les données exprimées en Json. Dans le cadre de travaux futurs, il serait intéressant d'étudier comment l'annotation des types pourrait être utilisée pour résoudre l'hétérogénéité entre le schéma de différents ensembles de données. Cela peut être envisagé en identifiant des types ayant la même sémantique en comparant leurs ensembles respectifs d'annotations. Une perspective intéressante serait d'investiguer la problématique de correspondance des propriétés des différentes sources de données.

Chapitre 6

Analyse et amélioration de la conformité entre une source de données RDF et son schéma

6.1 Introduction

Dans un jeu de données RDF, des déclarations sur le schéma peuvent être fournies avec les données. L'un des principaux avantages des langages du Web sémantique est qu'ils ne sont pas contraignants, c'est à dire que les données n'ont pas à suivre strictement les déclarations sur le schéma. Pour ces raisons, on peut retrouver des propriétés utilisées pour décrire les données et non déclarées dans le schéma, comme on peut retrouver des propriétés déclarées dans le schéma et non utilisées pour décrire les données. En plus du fait que ces langages n'imposent aucune contrainte sur la structure des données, même si des informations sur le schéma sont fournies explicitement dans le jeu de données, ce dernier peut évoluer et s'écarter des déclarations sur le schéma. Par conséquent, il existe généralement un écart entre les spécifications de schéma et les instances réelles dans le jeu de données.

Le problème de l'évaluation de l'écart existant entre un jeu de données et son schéma ne se pose pas pour le cas de données structurées qui suivent strictement un schéma. Afin d'exploiter des jeux de données RDF de façon pertinente, les applications et les utilisateurs ont besoin de leurs schémas, qui peuvent être fournis dans les jeux de données eux-mêmes. Cependant, l'analyse de jeux de données RDF réels, effectuée par [91], montre qu'ils sont rarement conformes à leur schéma : des instances de même type peuvent avoir des propriétés différentes, et les propriétés définies pour un type donné ne sont pas toujours les mêmes que celles spécifiées pour les instances de ce type. Une instance peut être décrite par la propriété *rdf*:*type*, qui définit son type. Deux instances ayant le même type ne sont pas nécessairement décrites par les mêmes propriétés. En outre, les jeux de données liées peuvent contenir des propriétés qui fournissent des informations sur le schéma, comme *rdfs*:*domain* et *rdfs*:*range*. Celles-ci peuvent être utilisées pour

déduire le schéma explicite du jeu de données.

Étant donnés une source et son schéma, et considérant qu'il peut exister un écart entre les deux, il serait intéressant de pouvoir évaluer cet écart afin de mettre en évidence la conformité entre un schéma et le jeu de données qu'il décrit. Cette conformité permet entre autre de donner une idée de la façon dont les propriétés des instances sont décrites au niveau du schéma, ce que nous désignons par pertinence du schéma ; ainsi qu'une idée du degré de renseignement des propriétés déclarées dans le schéma au niveau des instances, ce que nous désignons par complétude du jeu de données. En effet, une conformité faible peut refléter le fait que les données ont beaucoup évolué par rapport au schéma déclaré, et qu'il est utile de faire évoluer le schéma du jeu de données. Une valeur de conformité faible peut également indiquer que le jeu de données est très hétérogène, car des instances de même type sont décrites par des ensembles de propriétés différents. Dans ce cas, et si le schéma décrit bien toutes les propriétés des instances, il s'agit plutôt d'une limitation due aux langages utilisés (RDF(S)/OWL) qui ne permettent pas d'exprimer l'hétérogénéité des données au sein du même type.

L'évaluation de l'écart existant entre un jeu de données et son schéma permet également de savoir à quel point une propriété pertinente par rapport au besoin d'un utilisateur (par exemple : adresse), et qui figure dans le schéma, est réellement renseignée dans le jeu de données avant de l'interroger. Parfois, une propriété pertinente peut ne pas être décrite dans le schéma et être présente dans le jeu de données. Le problème abordé dans ce chapitre est l'évaluation de l'écart existant entre un jeu de données liées et son schéma afin que cela soit pris en considération pour l'exploitation de la source de données à travers son schéma.

Pour analyser l'écart qui existe entre un jeu de données et son schéma, nous proposons un ensemble de facteurs de qualité : (i) la complétude d'un jeu de données par rapport à son schéma, qui rend compte du degré auquel les propriétés déclarées dans le schéma sont renseignées au niveau des instances du jeu de données ; (ii) la pertinence d'un schéma par rapport à son jeu de donnée qui rend compte du degré de présence au niveau du schéma des propriétés décrivant les instances, et (iii) la conformité entre un jeu de données et son schéma qui reflète à la fois la complétude et la pertinence. Nous proposons également une extension du schéma qui permet de mieux refléter l'hétérogénéité des données. Comme les données ne respectent pas nécessairement le schéma déclaré, nous proposons également une approche qui permet de répercuter sur le schéma les changements survenant au niveau des données.

Ce chapitre est organisé de la manière suivante. Un exemple de motivation est présenté dans la section 6.2. Dans la section 6.3, nous décrivons la caractérisation de la source de données et du schéma. Les différents facteurs de qualité proposés sont ensuite présentés : la complétude d'une source de données est présentée dans la section 6.4 ; la pertinence du schéma est présentée dans la section 6.5 et la conformité entre une source de données et son schéma est présentée dans

la section 6.6. La section 6.7 présente nos propositions pour l'amélioration de la description d'une source de données. Nous présentons dans la section 6.8 quelques approches sur l'évaluation de la qualité des données du Web sémantique. La section 6.9 présente la méthodologie d'évaluation et les résultats obtenus en utilisant différents jeux de données. Nous concluons le chapitre en section 6.10.

6.2 Problèmes de conformité entre une source et son schéma

Une source de données comprend à la fois des instances ainsi que des informations relatives au schéma, comme : *rdf*:*type*, *rdfs*:*range*, *rdfs*:*domain*, etc. Les informations sur le schéma ne sont pas toujours fournies, et quand elles le sont, elles ne sont pas forcément en adéquation avec les données décrites. Pour évaluer l'écart entre une source de données et son schéma, les éléments suivants sont considérés :

- **Dans la source de données :** nous considérons la source de données en termes d'instances (voir définition dans chapitre 3 section 3.2.1) contenues dans la source. Nous ne considérons pas les propriétés primitives qui décrivent la structure des instances car elles sont plutôt considérées dans le schéma. Des propriétés issues de vocabulaires peuvent également être utilisées pour décrire les instances. Celles-ci sont considérées au même titre que les propriétés utilisateurs.
- **Dans le schéma :** Nous entendons par schéma de la source de données : (i) l'ensemble des types (qui ne sont pas des super-types) figurant dans la source comme objet d'une déclaration *rdf*:*type*, et (ii) les liens sémantiques entre ces types déclarés comme : "*p rdfs*:*range c*" et "*p rdfs*:*domain c*". Si une propriété d'un vocabulaire est utilisée pour décrire une instance, nous la considérons au même titre qu'une propriété non primitive pour laquelle les déclarations sur le domaine et le co-domaine sont fournies dans le schéma. Dans le cas d'une découverte du schéma implicite de la source de données, ces déclarations peuvent être générées de façon automatique.

Notons qu'il existe d'autres déclarations qui concernent le schéma telles que : *rdfs*:*subClassOf*, *owl*:*InverseFunctionalProperty*, etc. Cependant, dans ce chapitre nous nous focalisons sur les déclarations qui décrivent le mieux le jeu de données en termes de types et de liens sémantiques entre eux. L'investigation d'autres facette de la conformité par rapport à d'autres déclarations sur le schéma fera l'objet de nos travaux futurs.

La figure 6.1 montre l'exemple d'un jeu de données extrait de la *BNF*[1], qui contient des données de la Bibliothèque Nationale de France. Le schéma déclaré

1. BNF : datahub.io/fr/dataset/data-bnf-fr

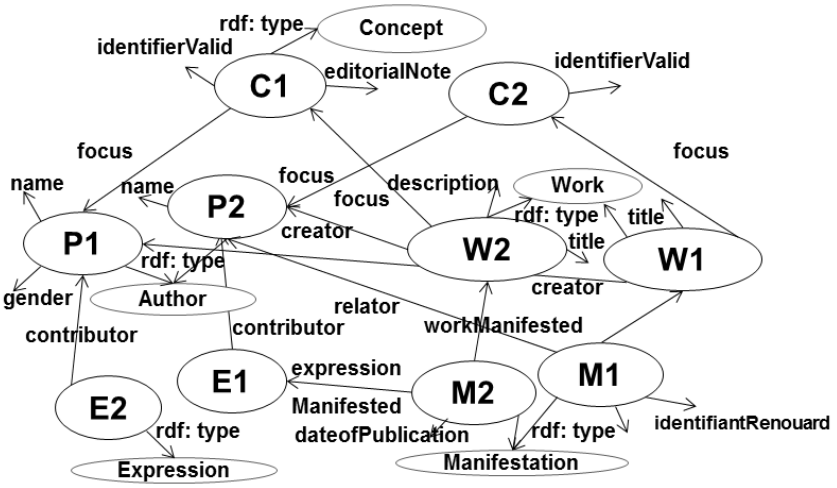


FIGURE 6.1 – Graphe de données.

pour ce jeu de données est représenté dans la figure 6.2. Nous pouvons voir que certaines instances sont décrites par la propriété *rdf:type*, définissant les types auxquels elles appartiennent. Pour d'autres instances, telles que $C2$ et $E1$, cette information est manquante. Deux instances ayant le même type ne sont pas nécessairement décrites par les mêmes propriétés, comme nous pouvons le voir pour $W1$ et $W2$ dans notre exemple. Elles sont toutes les deux associées au type *Work*, mais contrairement à $W1$, $W2$ possède la propriété *Description*.

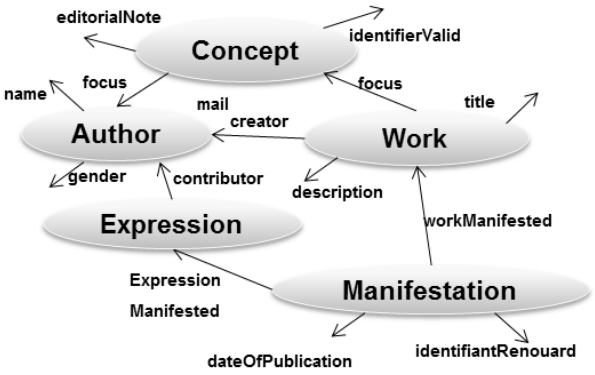


FIGURE 6.2 – Le schéma déclaré dans le jeu de données de la figure 6.1.

Notre objectif est d'évaluer la conformité entre une source de données et son schéma, et de fournir un ensemble de facteurs ainsi que les métriques associées pour capturer deux cas de non conformité : (i) les informations déclarées dans le schéma et non définies pour les instances de la source de données, et (ii) les informations définies pour des instances dans la source de données, mais non déclarées dans le schéma. Dans la figure 6.3 qui est un extrait de la source décrite

par les figures 6.1 et 6.2, les deux instances $P1$ et $P2$ sont définies comme ayant le type *Author*. Cependant, le type *Author* possède la propriété *gender* qui est définie pour $P1$ mais pas pour $P2$. Dans la figure 6.4 qui est également un extrait de la source décrite dans les figures 6.1 et 6.2, les deux instances $M1$ et $M2$ sont définies comme instances du type *Manifestation* ; mais l'instance $M1$ possède une propriété *relator* qui n'est pas déclarée comme une propriété du type *Manifestation* dans le schéma.

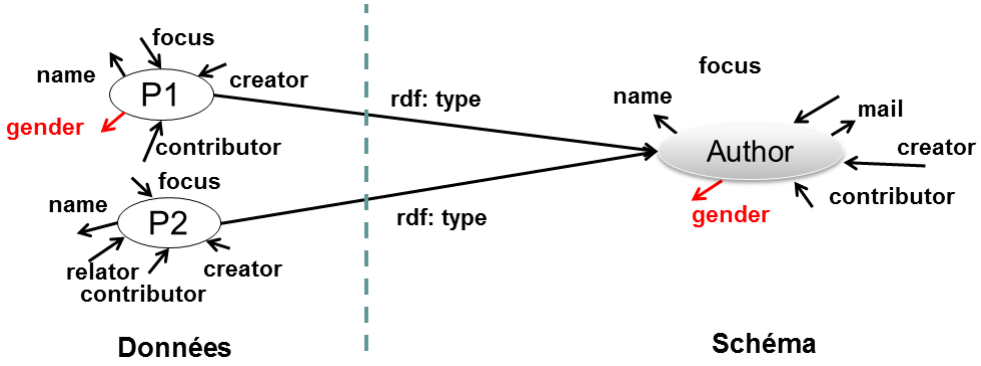


FIGURE 6.3 – Informations déclarées dans le schéma et non définies pour des instances d'une source de données.

Afin de détecter ces cas de non conformité, nous construisons le profil d'un type, décrivant les instances de la source de données, et d'autre part, pour chaque type déclaré, on dispose d'un ensemble de propriétés déclarées dans le schéma pour ce type. Nous utilisons le profil et l'ensemble de propriétés d'un type dans la définition de nos facteurs de qualité et des métriques associées pour évaluer l'écart entre les données et le schéma qui les décrit. Le processus de construction du profil et la prise en compte de l'ensemble de propriétés d'un type sont décrits dans les sections suivantes.

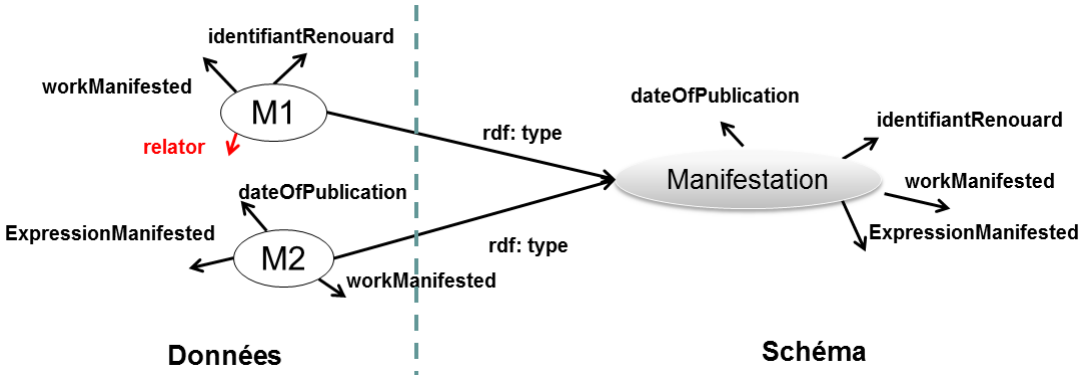


FIGURE 6.4 – Informations définies pour des instances d'une source de données et non déclarées dans le schéma.

Pour estimer dans quelle mesure les instances d'un type ont des valeurs pour les propriétés définies pour ce type dans le schéma, nous proposons le facteur de qualité de complétude de la source de données par rapport à son schéma. Pour estimer dans quelle mesure les propriétés des instances dans la source de données sont bien spécifiées comme propriétés des types du schéma, nous proposons le facteur de qualité de pertinence d'un schéma par rapport à la source de données décrite. Plus un schéma s'écarte des données qu'il décrit, moins ce dernier est pertinent. Enfin, nous introduisons une notion de conformité entre la source de données et son schéma, qui reflète à la fois la complétude et la pertinence.

6.3 Caractérisation de la source et de son schéma

Afin d'analyser la conformité entre un jeu de données et son schéma, nous construisons le profil d'un type pour caractériser le jeu de données. Cette notion de profil est identique à celle présentée dans le chapitre 3 pour caractériser les instances d'un type. Nous considérons également l'ensemble de propriétés d'un type, décrivant les propriétés déclarées dans le schéma pour un type donné. Nous les utilisons dans la définition de nos facteurs de qualité et des métriques associées pour évaluer l'écart entre les données et le schéma qui les décrit.

Dans ce qui suit, nous considérons les notations suivantes. Soient une source de données RDF notée D, l'ensemble $\Gamma = \{T_i : i = 1, ...n\}$ représente les types déclarés de D, et $\Phi = \{TP_i : i = 1, ...n\}$ représente l'ensemble des profils de types, où chaque type T_i est associé à un profil TP_i. On note $\rho = \{P_i : i = 1, ...n$ $\}$ l'ensemble des ensembles des propriétés de chaque type, où chaque type T_i est associé à un ensemble de propriétés P_i déclaré dans le schéma.

Le processus de construction des profils des types est décrit dans la section 6.3.1. La prise en compte des ensembles de propriétés déclarées pour un type est présentée dans la section 6.3.2.

6.3.1 Construction du profil de type

Contrairement à la découverte implicite des types présentée dans le chapitre 3, nous définissons, dans ce chapitre, un type comme un groupe d'instances ayant la même valeur pour la propriété *rdf:type* qui est déclarée explicitement. Notons que les super-types ne sont pas considérés dans ce chapitre.

Un profil de type est composé de l'ensemble des propriétés décrivant les instances du type. A chaque propriété est associée une probabilité. Une instance e peut avoir plusieurs types, et par-conséquent, elle peut introduire des propriétés dans le profil de type qui caractérisent d'autres types. Soit par exemple, une instance e de types *Author* et *Organizer*, avec e décrite par les propretés *name*, *made*, *affiliation* et *write* ; nous ne pouvons pas déterminer de façon automatique

quelle propriété appartient à quel type ; si par exemple l'occurrence de la propriété *made* est considérée dans le profil du type *Author*, alors cela aura introduit dans ce profil une propriété qui caractérise le type *Organizer*. Pour cela, nous proposons d'ajouter une propriété d'une instance e au profil d'un type T si et seulement si cette instance a uniquement le type T.

Un profil TP d'un type T dans la source de données D est composé d'un ensemble de propriétés utilisateurs. Chaque propriété p est associée à une probabilité α et annotée par une flèche indiquant sa direction. Nous construisons un profil TP pour chaque type T déclaré dans la source de données D. Pour chaque type T nous interrogeons la source pour extraire l'ensemble des propriétés décrivant une instance ayant pour seul le type T à l'aide des requêtes suivantes :

- $\forall p \in$ {Select ?p, COUNT(?t) as ?nbType Where { ?e rdf:type T . ?e ?p ?v . ?e rdf:type ?t}} : si ?nbType = 1 alors ajouter $(\overrightarrow{p}, \alpha) \in TP$;
- $\forall p \in$ {Select ?p, COUNT(?t) as ?nbType Where { ?e rdf:type T . ?v ?p ?e . ?e rdf:type ?t}} : si ?nbType = 1 alors ajouter $(\overleftarrow{p}, \alpha) \in TP$.

Dans un profil de type $TP = \{(p_1, \alpha_1), ..., (p_n, \alpha_n)\}$, chaque p_i représente une propriété et chaque α_i représente la probabilité pour une instance de T d'être décrite par la propriété p_i. La probabilité α_i associée à une propriété p_i dans le profil du type T est évaluée comme le nombre d'instances du type T pour lesquelles p_i est définie sur le nombre total d'instances dans T. Le nombre d'instances du type T pour lesquelles p_i est définie est retrouvé en interrogeant la source à l'aide des requêtes suivantes :

- Select COUNT(distinct(?e)) as ?nbEntitiesP Where { ?e rdf:type T . ?e p_i ?v } ; Affecter une probabilité de $\overrightarrow{p_i}$: $(\overrightarrow{p_i}, ?nbEntitiesP/|T|) \in TP$;
- Select COUNT(distinct(?e)) as ?nbEntitiesP Where { ?e rdf:type T . ?v p_i ?e } ; Affecter une probabilité de $\overleftarrow{p_i}$: $(\overleftarrow{p_i}, ?nbEntitiesP/|T|) \in TP$.
- Avec |T| = Select COUNT(distinct(?e)) Where { ?e rdf:type T) }

Les profils des types de l'exemple donné dans la figure 6.1 sont représentés dans la table 6.1. Dans le profil du type *Author*, la probabilité de la propriété *gender* est de 0.5 car $P1$ possède cette propriété alors que $P2$ ne la possède pas.

La différence entre les profils de type construits lors du processus de découverte de schéma et ceux utilisés ici réside dans la façon dont ils sont obtenus : dans ce chapitre, nous construisons ces profils à l'aide des déclarations *rdf:type* présentes dans la source de données.

6.3.2 Ensemble de propriétés déclarées pour un type

En plus du profil de type, nous considérons l'ensemble de propriétés de chaque type défini dans le schéma de la source de données. Un ensemble de propriétés d'un type est composé des propriétés déclarées dans le schéma ayant

TABLE 6.1 – Profils des types de la source de données de la figure 6.1.

Type	Profil de Type
Concept	$\{(\overrightarrow{editorialNote}, 0.5), (\overrightarrow{identifierValid}, 1), (\overrightarrow{focus}, 1), (\overleftarrow{focus}, 1)\}$
Author	$\{(\overrightarrow{name}, 1), (\overleftarrow{creator}, 1), (\overleftarrow{relator}, 0.5), (\overleftarrow{focus}, 1), (\overrightarrow{gender}, 0.5), (\overleftarrow{contributor}, 1)\}$
Work	$\{(\overrightarrow{title}, 1), (\overleftarrow{workManifested}, 1)(\overrightarrow{creator}, 1), (\overrightarrow{description}, 0.5), (\overrightarrow{focus}, 1)\}$
Manifestation	$\{(\overrightarrow{workManifested}, 1), (\overrightarrow{dateOfPublication}, 0.5), (\overrightarrow{identifiantRenouard}, 0.5), (\overrightarrow{expressionManifested}, 0.5),(\overrightarrow{relator}, 0.5)\}$
Expression	$\{(\overrightarrow{contributor}, 1), (\overleftarrow{expressionManifested}, 0.5)\}$

comme domaine ou co-domaine ce type. L'ensemble de propriétés déclarées pour un type est identifié comme suit.

Un type T, dans le schéma du jeu de données D est décrit par son ensemble de propriétés P. Chaque propriété p est annotée par une flèche indiquant son orientation, et elle est telle que :

- Si $\exists(p, rdfs{:}domain, T) \in D$ alors $\overrightarrow{p} \in P$;
- Si $\exists(p, rdfs{:}range, T) \in D$ alors $\overleftarrow{p} \in P$.

L'ensemble de propriétés d'un type est identifié pour chaque type dans le schéma, en interrogeant la source de données D à l'aide des requêtes suivantes :

- $\forall p \in$ (Select ?p Where { ?p rdfs:domain T }), ajouter $\overrightarrow{p} \in P$;
- $\forall p \in$ (Select ?p Where { ?p rdfs:range T }), ajouter $\overleftarrow{p} \in P$.

Les ensembles de propriétés des types pour le schéma de la figure 6.2 sont représentés dans la table 6.2. La propriété *mail* appartient à l'ensemble des propriétés déclarées pour le type *Author*, mais pas au profil du type *Author* car elle n'est définie pour aucune instance de ce type dans la source de données. La propriété *relator* n'est pas dans l'ensemble de propriétés déclarées pour le type *Manifestation* car elle n'est pas déclarée dans le schéma. Cependant, cette propriété appartient au profil du type *Manifestation* (voir la table 6.1) car elle est définie pour certaines de ses instances.

TABLE 6.2 – Les ensembles de propriétés des types de la figure 6.2.

Type	Ensemble de propriétés
Concept	$\{\vec{editorialNote}, \vec{identifierValid}, \overleftarrow{focus}, \vec{focus}\}$
Author	$\{\vec{name}, \overleftarrow{creator}, \vec{mail}, \overleftarrow{focus}, \vec{gender}, \overleftarrow{contributor}\}$
Work	$\{\vec{title}, \overleftarrow{workManifested}, \vec{creator}, \vec{description}, \vec{focus}\}$
Manifestation	$\{\vec{identifiantRenouard}, \vec{expressionManifested}, \vec{dateOfPublication}, \vec{workManifested}\}$
Expression	$\{\vec{contributor}, \overleftarrow{expressionManifested}\}$

6.4 Complétude d'une source de données

La complétude d'une source de données exprime dans quelle mesure une propriété définie pour un type dans le schéma est renseignée pour les instances de ce type. Elle peut être définie à trois niveaux de granularité différents : la propriété, le type ou la source de données.

Complétude d'une propriété. La complétude d'une propriété est un facteur qui rend compte de la façon dont une propriété déclarée pour un type dans le schéma, est renseignée pour les instances de ce type.

La complétude d'une propriété est évaluée en comparant un profil de type à son ensemble de propriétés déclarées dans le schéma. Pour chaque propriété du type, la complétude est la probabilité de cette propriété dans le profil de type lorsqu'elle y figure. Elle est de 0 si la propriété n'appartient pas au profil du type. Dans notre exemple donné en figure 6.1, pour le type *Author*, si on compare le profil de type dans la table 6.1 avec son ensemble de propriétés dans la table 6.2, la complétude de la propriété *gender* est de 0.5 car la probabilité de cette propriété est de 0.5 dans le profil du type. La complétude de la propriété *mail* est de 0 car cette propriété est déclarée pour le type mais elle ne figure pas dans le profil de type, ce qui signifie qu'aucune instance de ce type n'est caractérisée par cette propriété.

Pour évaluer la complétude de la source de données par rapport à son schéma, nous comparons les propriétés de chaque profil de type TP_i avec les propriétés de son ensemble de propriétés P_i. Nous utilisons la probabilité des propriétés dans TP_i pour évaluer la complétude d'une propriété p dans les instances de la source de données, comme décrit dans la formule 6.1.

$$\forall p \in P_i, ComplétudePropriété(p) = \begin{cases} \alpha & \text{si} \quad p \in TP_i \\ 0 & \text{sinon} \end{cases} \tag{6.1}$$

Dans la formule ci-dessus, α représente la probabilité associée à la propriété p dans TP_i.

Complétude d'un profil de type. La complétude d'un type est un facteur qui rend compte de la façon dont les propriétés déclarées pour un type dans le schéma, sont renseignées pour les instances de ce type.

Nous évaluons la complétude d'un type T_i comme la valeur moyenne des complétudes des propriétés de TP_i, comme le montre la formule 6.2.

$$ComplétudeType(T_i) = \frac{\sum_{\forall p \in P_i} ComplétudePropriété(p)}{|P_i|} \tag{6.2}$$

Complétude de la source. La complétude d'une source de données est un facteur qui rend compte de la façon dont le schéma déclaré pour une source de données est renseigné pour les instances de cette source.

Nous évaluons la complétude de la source de données D par rapport à son schéma S comme la moyenne des valeurs de complétude de l'ensemble des types Γ du jeu de données, comme le montre la formule 6.3.

$$ComplétudeSource(D) = \frac{\sum_{\forall T_i \in \Gamma} ComplétudeType(T_i)}{|\Gamma|} \tag{6.3}$$

6.5 Pertinence du schéma

La pertinence du schéma exprime dans quelle mesure les propriétés des instances d'un type sont définies dans le schéma. Une propriété est définie dans le schéma à travers les déclarations explicites *rdfs:domain* et *rdfs:range*. Comme pour la complétude, la pertinence est définie à trois niveaux de granularité différents : la propriété, le type ou le schéma.

Pertinence d'une propriété. La pertinence d'une propriété pour un type dans un schéma rend compte de la présence d'une propriété décrivant les instances d'un type dans les déclarations du schéma.

La pertinence d'une propriété est évaluée en comparant la propriété considérée dans le profil de type par rapport à la déclaration dans le schéma. Pour chaque propriété du profil, la pertinence est la probabilité de cette propriété dans le profil de type, ou le complément de cette probabilité si la propriété n'est pas déclarée dans le schéma. Dans notre exemple donné en figure 6.1, considérons le type *Author* ; si on compare le profil associé dans la table 6.1 avec l'ensemble de propriétés du type dans la table 6.2, on obtient une valeur de 1 pour la pertinence de la propriété *name* car la probabilité de cette propriété est de 1 dans le profil et cette propriété est bien déclarée dans le schéma. La pertinence de la propriété

relator est de 0.6 ce qui représente le complément de sa probabilité dans le profil de type (1 - 0.4), car cette propriété n'est pas déclarée dans le schéma.

Pour évaluer la pertinence du schéma pour une source de données, nous comparons les propriétés de chaque ensemble de propriétés de types P_i au profil du type correspondant TP_i. Nous considérons la probabilité des propriétés dans TP_i pour évaluer la pertinence d'une propriété p dans le schéma, comme le montre la formule 6.4.

$$\forall p \in TP_i, PertinencePropriété(p) = \begin{cases} \alpha & \text{si} \quad p \in P_i \\ 1-\alpha & \text{sinon} \end{cases} \tag{6.4}$$

Dans la formule ci-dessus, α représente la probabilité associée à la propriété p dans TP_i.

Pertinence d'un type. La pertinence d'un type rend compte du degré de présence des propriétés décrivant les instances d'un type, dans les déclarations du schéma associé.

Nous évaluons la pertinence d'un type T_i par rapport à son ensemble de propriétés P_i déclarées dans le schéma, comme la valeur moyenne de pertinence des propriétés dans le profil de type TP_i, comme le montre la formule 6.5.

$$PertinenceType(T_i) = \frac{\sum_{\forall p \in TP_i} PertinencePropriété(p)}{|TP_i|} \tag{6.5}$$

Pertinence d'un schéma. La pertinence d'un schéma rend compte du degré de présence des propriétés caractérisant les instances de la source au niveau du schéma.

Nous évaluons la pertinence d'un schéma S par rapport à la source de données correspondante D comme la valeur moyenne de pertinence de l'ensemble des types Γ, comme cela est décrit dans la formule 6.6.

$$PertinenceSchéma(S) = \frac{\sum_{\forall T_i \in \Gamma} PertinenceType(T_i)}{|\Gamma|} \tag{6.6}$$

6.6 Conformité entre une source de données et son schéma

Nous définissons la conformité entre une source de données et son schéma, comme un facteur de qualité qui reflète à la fois la complétude de la source et la pertinence du schéma. L'idée est de rendre compte d'une façon générale des différences entre ce qui est spécifié dans le schéma et ce qui existe dans les instances de la source de données. Elle exprime dans quelle mesure un schéma

correspond bien à la source de données qu'il décrit. Elle est définie à trois niveaux de granularité différents : au niveau de la propriété, au niveau d'un type, ou entre la source de données et le schéma correspondant.

Conformité d'une propriété. La conformité d'une propriété est un facteur qui rend compte à la fois de la façon dont une propriété déclarée pour un type dans le schéma est renseignée pour les instances de ce type, et de la présence d'une propriété décrivant les instances d'un type dans les déclarations du schéma.

La conformité d'une propriété est évaluée en comparant l'ensemble de propriétés déclarées pour un type et son profil. Pour chaque propriété de l'ensemble de propriétés ou du profil, la conformité est de 0 si la propriété n'est pas dans le profil du type. Elle est égale à la probabilité de la propriété dans le profil du type si elle est présente à la fois dans le profil de type et dans l'ensemble de propriétés. La conformité d'une propriété est le complément de sa probabilité dans le profil de type si elle n'est pas déclarée dans le schéma pour ce type.

Dans notre exemple de la figure 6.1, si on compare le profil du type *Author* dans la table 6.1 avec son ensemble de propriétés dans la table 6.2, on trouve que la conformité de la propriété *name* est de 1 car la probabilité de cette propriété est de 1 dans le profil de type et que cette propriété est bien déclarée pour ce type. La conformité de la propriété *relator* est de 0.6, ce qui représente le complément de sa probabilité dans le profil de type (1 - 0.4), car cette propriété n'est pas déclarée dans le schéma. La conformité de la propriété *mail* est de 0 car cette propriété est déclarée dans le schéma pour ce type mais elle ne figure pas dans le profil du type car elle ne décrit aucune de ses instances.

Pour évaluer la conformité entre une source de données et son schéma, nous évaluons d'abord la conformité d'une propriété p entre un profil de type TP_i son ensemble de propriétés P_i, comme décrit dans la formule 6.7.

$$\forall p \in (TP_i \cup P_i), ConformitéPropriété(p) = \begin{cases} 0 & \text{si} \quad p \notin TP_i \wedge p \in P_i \\ \alpha & \text{si} \quad p \in TP_i \cap P_i \\ 1-\alpha & \text{si} \quad p \in TP_i \wedge p \notin P_i \end{cases} \tag{6.7}$$

Dans la formule ci-dessus, α représente la probabilité associée à la propriété p dans TP_i.

Conformité entre un profil de type et son ensemble de propriétés. La conformité entre un profil de type et son ensemble de propriétés est un facteur qui rend compte à la fois de la façon dont les propriétés déclarées pour un type dans le schéma sont renseignées pour les instances de ce type, et de la présence des propriétés décrivant les instances d'un type dans les déclarations du schéma associé.

Nous évaluons la conformité d'un type T_i dans la source de données par rap-

port à son schéma comme la moyenne des valeurs de la conformité de ses propriétés. Étant donné un profil de type TP_i et son ensemble de propriétés P_i, nous calculons leur conformité comme le montre la formule 6.8.

$$ConformitéType(T_i) = \frac{\sum_{\forall p \in (TP_i \cup P_i)} ConformitéPropriété(p)}{\sum_{\forall p \in (TP_i \cup P_i)} 1} \tag{6.8}$$

En utilisant la complétude d'un profil de type par rapport à ses instances dans la source de données et la pertinence d'un ensemble de propriétés dans le schéma, la conformité d'un type est définie sur la base de sa pertinence et de sa complétude comme suit :

$$ConformitéType(T_i) = Min\{ComplétudeType(T_i), PertinenceType(T_i)\}$$

Conformité entre une source de données et son schéma. La conformité entre une source de données et son schéma est un facteur qui rend compte à la fois de la façon dont le schéma déclaré pour une source de données est renseigné dans les instances de cette source, et du degré de présence des propriétés caractérisant les instances de la source au niveau du schéma.

De façon plus générale, nous évaluons la conformité entre une source de données D et son schéma S, comme la valeur moyenne de la conformité entre l'ensemble des types Γ. La formule correspondante est décrite dans la formule 6.9.

$$Conformité(D, S) = \frac{\sum_{\forall T_i \in \Gamma} ConformitéType(T_i)}{|\Gamma|} \tag{6.9}$$

6.7 Amélioration de la description d'une source de données

La disparité existante entre les jeux de données et leurs schémas est une conséquence logique de la flexibilité des langages utilisés pour exprimer les données liées. En effet, ces langages n'imposent aucune contrainte sur la structure des données ce qui creuse un écart évident entre le schéma déclaré et la réalité de la structure des données insérées. Un problème ouvert est l'amélioration du schéma d'un jeu de données afin de réduire cette disparité, et de garantir l'évolution du schéma avec les données qu'il décrit.

Nous proposons dans ce qui suit une extension du schéma qui permet de mieux décrire un jeu de données ainsi qu'une étude sur l'évolution du schéma suite aux évolutions survenant dans les données. Comme nous ne pouvons pas altérer les données, nos améliorations sont centrées sur le schéma uniquement.

6.7.1 Extension du schéma

Un raisonneur tel que Jena peut compléter certaines déclarations manquantes du schéma, par exemple sur le domaine et le co-domaine des propriétés. Il serait par conséquent préférable d'utiliser d'abord ce type de raisonneur pour tenter de réduire l'écart entre une source de données et son schéma. Cependant, un raisonneur ne peut pas refléter dans le schéma l'hétérogénéité des instances de même type. Nous proposons de réduire l'écart entre un jeu de données et son schéma en introduisant une extension du schéma qui permet de refléter cette hétérogénéité. En plus de décrire une propriété par son domaine et son co-domaine, nous proposons de lui associer des probabilités.

En effet, pour réduire l'écart entre un jeu de donnée et son schéma, il faut d'abord améliorer la description de la plus petite granularité du jeu de données, qui est la propriété. Une propriété est décrite dans le schéma par un domaine et un co-domaine dont les valeurs sont des types, comme décrit dans la figure 6.5 (a). Cependant, une propriété peut servir à décrire seulement certaines instances d'un type. Ce qui nous a conduit à introduire des probabilité d'appartenance des propriétés dans les profils de type. Ces probabilités donnent une idée de la présence d'une propriété au niveau d'un type et ainsi son importance pour refléter le type.

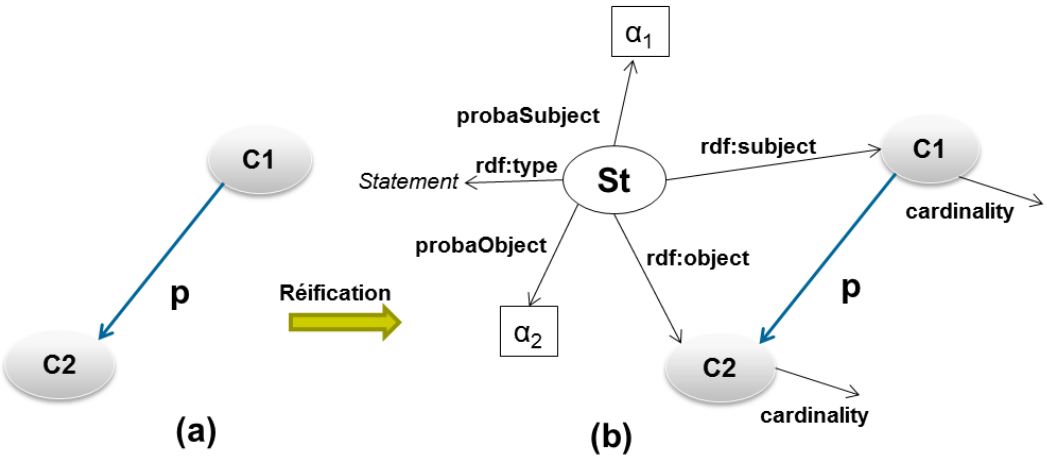


FIGURE 6.5 – Extension du schéma

Définition (Probabilités d'une propriété). Une propriété peut être entrante pour des instances d'un type $c1$ et sortante pour des instances d'un type $c2$. Par conséquent deux probabilités peuvent être attribuées :

- **probaSubject**, qui est la probabilité qu'une instance de type $c1$ soit décrite par p ;
- **probaObject**, qui est la probabilité qu'une instance de type $c2$ d'être en objet de la propriété p.

Nous proposons d'introduire ces probabilités au niveau du schéma en utilisant la réification sur les déclarations des propriétés définies, comme le montre la figure 6.5 (b). La réification [14] est une description d'une déclaration en RDF. Le vocabulaire de réification RDF se compose du type *rdf*:*Statement*, et des propriétés *rdf*:*subject*, *rdf*:*predicate*, et *rdf*:*object*. Nous définissons la déclaration des probabilités d'une propriété en utilisant la réification comme suit.

Définition (Description des probabilités d'une propriété). Soit une propriété p dont le domaine est le type c_1 et dont le co-domaine est le type c_2. Soient α_1 la probabilité d'une instance de c_1 d'être décrite par la propriété $\overrightarrow{p}$ et α_2 la probabilité d'une instance de c_2 d'être décrite par la propriété $\overleftarrow{p}$. Nous proposons de décrire la propriété p entre les instances des types c_1 et c_2 comme suit :

- *st* **rdf:type** *rdf*:*Statement*
- *st* **rdf:subject** c_1
- *st* **rdf:predicate** p
- *st* **rdf:object** c_2
- *st* **probaSubject** α_1
- *st* **probaObject** α_2

Des instances d'un type donné peuvent être décrites par des propriétés non déclarées dans le schéma. Dans ce cas, il suffit d'ajouter les déclarations de ces propriétés dans le schéma comme décrit dans la figure 6.5 (b) et dans la définition ci-dessus.

Un autre cas possible est celui où des propriétés sont déclarées dans le schéma avec un domaine et un co-domaine, sans qu'aucune instance dans le jeu de données ne soit décrites par ces propriétés. Dans ce cas, les probabilités de ces propriétés sont mises à 0 dans le schéma pour qu'il soit conforme au jeu de données.

Certains types dans un jeu de données peuvent être définis pour certaines instances et ne pas être déclarés dans le schéma comme une classe. Dans ce cas, la déclaration d'un type T est ajoutée dans le schéma comme suit : T *rdf*:*type class*.

Un autre cas possible est celui d'un type déclaré dans le schéma et pour lequel il n'y a aucune instance dans le jeu de données. Il serait intéressant que le schéma puisse refléter le nombre d'instances d'un type. Nous proposons pour cela d'attribuer à chaque type dans le schéma une propriété notée *cardinality* ayant comme valeur le nombre d'instances d'un type dans le jeu de données, comme le montre la figure 6.5.

6.7.2 Évolution du schéma

Pour que le schéma reflète à chaque instant le contenu du jeu de données, il est nécessaire de le faire évoluer à chaque modification des données. Nous considérons dans cette section un schéma avec l'extension proposée en section 6.7.1. Les modifications affectant le schéma concernent en particulier les insertions et suppressions d'instances. En effet, l'insertion/suppression d'une instance impacte le schéma et plus précisément les probabilités du sujet et de l'objet des propriétés de cette instance. La modification d'une instance peut également affecter le schéma, si cette modification concerne l'insertion d'une nouvelle propriété pour une instance ou bien sa suppression. En revanche, si la modification de l'instance concerne uniquement les valeurs des propriétés, cela n'affecte pas le schéma.

Dans ce qui suit, nous présentons les modifications à apporter au schéma dans le cas de l'insertion/suppression d'une propriété (section 6.7.2.1), et dans le cas de l'insertion/suppression d'une instance (section 6.7.2.2).

6.7.2.1 Insertion/Suppression d'une propriété

A l'insertion ou à la suppression d'une propriété décrivant une instance, il faut mettre à jour les probabilités du sujet et de l'objet de cette propriété. Pour mettre à jour ces probabilités, nous avons besoins de connaître les probabilités actuelles de cette propriété dans le schéma. Les probabilités du sujet et de l'objet d'une propriété p sont obtenues par une requête de la forme :

"Select ?*probaSubject*, ?*probaObject* Where {
?st rdf:subject ?c_1 .
?st rdf:predicate p .
?st rdf:object ?c_2 .
?st probaSubject ?*probaSubject* .
?st probaObject ?*probaObject*
}"

Une autre valeur importante pour recalculer les nouvelles probabilités est le nombre d'instances d'un type. Cette information est disponible à travers l'extension du schéma proposée qui permet également d'attribuer à un type dans le schéma une propriété (*cardinality*) ayant comme valeur le nombre d'instances d'un type dans le jeu de données. La probabilité α d'une propriété insérée (+) ou supprimée (-) pour un type T_i est calculée comme suit :

$$\alpha = \frac{(\alpha \times nbInstancesType)(+/-)1}{nbInstancesType(+/-)1} \tag{6.10}$$

6.7.2.2 Insertion/Suppression d'une instance

A l'insertion ou à la suppression d'une instance, il faudrait mettre à jour les probabilités des propriétés décrivant cette instance comme décrit dans la section 6.7.2.1, ainsi que la cardinalité de ses types. En effet, la suppression d'une instance engendre la suppression de toutes ses propriétés, comme l'insertion d'une instance engendre une nouvelle occurrence de toutes les propriétés la décrivant.

L'insertion d'une instance peut provoquer l'apparition d'un nouveau type si son type n'est pas déjà déclaré dans le schéma. Nous proposons dans ce cas d'ajouter un nouveau type dans le schéma pour être conforme au jeu de données, et également d'associer les propriétés de cette instance à ce nouveau type.

6.8 Approches sur la qualité des données liées du Web sémantique

Plusieurs approches portant sur la qualité des données du Web sémantique ont été proposées dans la littérature. Cependant, à notre connaissance, il n'existe pas de travaux qui abordent la conformité entre un jeu de données et son schéma. Nous présentons dans cette section d'une manière non exhaustive certaines approches qui abordent le problème de l'évaluation de la qualité des données du Web sémantique, et des approches qui concernent l'amélioration de la qualité de ces données.

6.8.1 Évaluation de la qualité des données

De nombreux travaux de recherche ont abordé le thème de la qualité des données, une revue de ces travaux est proposée dans [92]. La qualité est décrite à travers un grand ensemble de facteurs dans [93, 94]. Des modèles de qualité [95], ainsi que des facteurs et des métriques ont été proposés [96], ainsi que des approches pour évaluer ces facteurs [97].

La qualité de l'information dans le contexte spécifique du Web sémantique a été adressée par plusieurs travaux de recherche. Les auteurs de [98] proposent un ensemble de requêtes SPARQL génériques pour identifier les valeurs illégales des littéraux ainsi que les violations de dépendances fonctionnelles. Furber et al. [99] montrent comment les langages d'interrogation SPARQL et SPARQL inferences Notation (SPIN) peuvent être utilisés pour identifier les problèmes de qualité de données. Un modèle conceptuel est présenté dans [100], pour permettre la gestion de la qualité des données sous la forme d'une ontologie permettant la formulation de règles standardisées de qualité et de nettoyage de données ainsi que la classification des problèmes de qualité de données et le calcul de scores de qualité pour les sources de données du Web sémantique.

Plusieurs outils d'évaluation de la qualité pour les données du Web sémantique ont été proposés. SWIQA [101] est un outil qui classe les problèmes de qualité de données et calcule des scores de qualité pour différentes dimensions de qualité : unicité par rapport aux propriété définies comme uniques, complétude par rapport aux propriété obligatoires, précision syntaxique par rapport aux valeurs légales des propriétés, précision sémantique par rapport aux dépendances fonctionnelles, etc. Sieve [102] est un framework flexible pour exprimer à la fois l'évaluation de la qualité et de la fusion. Il représente un composant de l'outil LDIF [103] qui est proposé pour l'intégration des données liées. Il gère l'accès aux données, la cartographie de schéma et la résolution d'entités, ainsi que l'évaluation de la qualité et la fusion. LDIF est capable de fusionner les différents identifiants d'un même objet en une URI canonique, et les attributs équivalents du schéma ainsi que les noms de classes en une représentation cible homogène. En conséquence d'un tel processus d'intégration de données, plusieurs valeurs pour le même attribut peuvent être observées, et Sieve permet de résoudre ces conflits.

Les travaux présentés dans [104, 105] proposent une méthodologie pour évaluer la qualité des ressources de données liées. La première phase comprend la détection de problèmes de qualité communs et leur représentation dans une taxonomie. La deuxième phase comprend l'évaluation d'un grand nombre de ressources individuelles par crowdsourcing en fonction du problème de qualité portant par exemple sur : la précision, l'interconnexion vers d'autres sources de données, la consistance, etc. L'approche proposée dans [106] présente une méthodologie pour l'évaluation de la qualité des données liées qui est pilotée par un ensemble de tests, ceci à l'aide d'un ensemble de modèles de requêtes SPARQL.

Certaines approches ont abordé le problème de l'évaluation de la fiabilité de l'information sur le Web en fonction de sa provenance, qui est essentielle pour évaluer la qualité et la fiabilité des informations sur le Web. Les auteurs de [107] présentent une approche qui utilise les informations de provenance pour évaluer la qualité des données RDF. Puis, dans [108], un modèle de confiance est proposé : il associe les déclarations RDF avec des valeurs de confiance et étend le langage SPARQL pour accéder à ces valeurs de confiance.

6.8.2 Amélioration de la qualité du schéma pour des sources RDF

Certaines approches ont été proposées pour améliorer la qualité du schéma pour les données du Web sémantique. Ainsi, l'approche SDValidate [109] ajoute des informations de type manquantes à une source de données. L'approche est proposée pour identifier des déclarations potentiellement erronées qui ont été générées par le système d'extraction d'information. Cette méthode propose d'utiliser la distribution statistique des types et des propriétés. Dans [110], le solveur IBM ILOG CPLEX est utilisé pour affiner les classes afin de parvenir à

une meilleure homogénéité de la structure. La source de données ainsi que les contraintes sont fournies au solveur qui renvoie le meilleur partitionnement. Les auteurs proposent également un cadre général pour permettre aux utilisateurs de définir leurs propres mesures de similarité structurelles personnalisées d'une manière simple. À cette fin, un langage pour décrire ces mesures est proposé, avec une syntaxe simple et une sémantique formelle. Cette approche propose également la recherche des co-occurrences de propriétés pour affiner la définition des classes : une classe est affinée en divisant ses instances en fonction de leur structure exacte. Les propriétés ayant une forte probabilité d'apparaître ensemble ont une forte probabilité d'être dans la même sous-classe. Contrairement à notre approche, cette méthode n'évalue pas la conformité de la source de données à son schéma. De plus, elle est très coûteuse et nécessite des machines puissantes : pour une machine de 64 Go de RAM, 6-core, l'exécution prend 75 heures pour la classe *Noun* de Wordnet [111].

6.8.3 Discussion et positionnement par rapport aux approches relatives à la qualité des données du Web sémantique

Plusieurs approches d'évaluation de la qualité de l'information sont proposées pour les données du Web sémantique. Ces approches abordent différents problèmes de la qualité de l'information, comme par exemple : les types de données manquants, les valeurs illégales des littéraux et les violations de dépendances fonctionnelles [98] ; les valeurs multiples d'un attribut [102] ; la fiabilité de l'information [107, 108]. Ces approches utilisent différentes façons, par exemple : des requêtes SPARQL génériques [98], un modèle conceptuel sous la forme d'une ontologie permettant la formulation de règles standardisées de qualité et de nettoyage de données [100], des techniques de *crowdsourcing* [104, 105], outil de fusion ou d'intégration de plusieurs sources [102], pilotage par un ensemble de tests [106] ou l'analyse de la provenance [107, 108].

La qualité de l'information est aussi bien évaluée pour caractériser une seule source qu'à des fins de fusion [102], ou d'intégration [103].

Notre approche complémente ces travaux en proposant de nouveaux facteurs de qualité et les métriques associées. Les travaux existants ont proposés un certain nombre de facteurs pour caractériser la qualité des données du Web, par exemple, le travail présenté dans [98] identifie les propriétés de types de données manquantes, les valeurs illégales ainsi que les violations de dépendances fonctionnelles. D'autre travaux présentés dans [104, 105] évaluent la qualité de l'information par *crowdsourcing* ; et le travail présenté dans [106] évalue la qualité des données par un ensemble de tests. Certaines approches [112, 107, 108] ont abordé le problème de l'évaluation de la fiabilité de l'information sur le Web en

fonction de leur provenance. Les facteurs de qualité que nous proposons peuvent également être considérés comme des facteurs de qualité supplémentaires dans des outils tels que SWIQA [101] ou Sieve [102].

Des pistes pour l'amélioration de la qualité de l'information ont été également adressées. Cela en identifiant des déclarations potentiellement erronées ou en complétant les informations manquantes sur les types [109], ou encore en étudiant la distribution statistique des types et des propriétés. L'approche présentée dans [110] adresse la qualité de description des données et propose de l'améliorer en partitionnant le jeu de données en classes d'instances structurellement homogènes. Contrairement à ces approches, notre proposition pour l'amélioration de la conformité entre un schéma et la source qu'il décrit ne consiste pas à créer de nouveaux types ou à restructurer des types existants, mais plutôt à proposer une extension du schéma qui permet de renseigner sur la conformité et ainsi mieux décrire la source de données.

6.9 Expérimentation

Cette section présente quelques résultats d'expérimentation sur l'évaluation de la conformité entre différents jeux de données réels et leur schéma.

6.9.1 Sources de données

Nous avons évalué la conformité du schéma de trois jeux de données réels :

- *Conference*[2], qui décrit des articles, des présentations et des personnes pour plusieurs conférences et ateliers du Web sémantique ;
- *BNF*[3], qui est un jeu de données qui contient des données sur la Bibliothèque Nationale de France ;
- Un jeu de données extrait de *DBpedia* [63], en considérant les types suivants : *Politician*, *Footballeur*, *Museum*, *Film*, *Book* et *Country*.

Nous avons choisi le jeu de données *Conference* car il ne dispose pas de déclarations sur le schéma. Pour permettre l'évaluation de la conformité, nous avons d'abord découvert le schéma implicite des données. Pour les jeux de données *BNF* et *DBpedia*, les déclarations sur le schéma sont fournies.

6.9.2 Méthodologie

Afin d'évaluer l'écart entre une source de données et son schéma, nous avons identifié dans le jeu de données les types feuilles dans la hiérarchie de types, c'est à dire, les types qui ne sont pas super-types d'un type existant, puis nous

2. Conférence : data.semanticweb.org/dumps/conferences/dc-2010-complete.rdf
3. BNF : datahub.io/fr/dataset/data-bnf-fr

avons construit leurs profils comme décrit dans la section 6.3.1. Nous avons également considéré l'ensemble de propriétés de chaque type à partir des déclarations *rdfs:range* et *rdfs:domain* sur le schéma dans le jeu de données, comme décrit dans la section 6.3.2. Notons que pour le jeu de données *Conference*, ces déclarations sont inexistantes, nous les avons découvertes et générées par notre approche de découverte du schéma implicite d'un jeu de données décrite dans le chapitre 2. Nous avons annoté chaque cluster (classe) découvert avec la déclaration du type la plus fréquente de ses instances. Si des instances d'un cluster n'ont pas de label pour leur type, nous les avons étiquetées comme non typées (*untyped*).

Pour chaque profil de type identifié TP_i, nous avons comparé ses propriétés avec celles de son ensemble de propriétés P_i déclarées dans le schéma. En utilisant les différents facteurs définis précédemment, nous considérons la probabilité des propriétés pour évaluer la complétude, la pertinence et la conformité d'une propriété p entre les instances d'un type. Puis nous évaluons la complétude, la pertinence et la conformité d'un type T_i. Enfin, nous évaluons la complétude de la source de données, la pertinence du schéma et la conformité entre le jeu de données et le schéma.

6.9.3 Résultats

La table 6.3 montre les profils des types du jeu de données *Conference*. Nous pouvons voir à travers les probabilités des propriétés qui sont égales à 1, que les instances de même type sont très homogènes. Cependant, les instances de type *Person* sont moins homogènes avec, par exemple, la propriété *made* qui a une probabilité de 0.53 et la propriété *heldBy* qui a une probabilité de 0.56. La découverte automatique du schéma implicite de ce jeu de données a montré qu'il existe des classes dont les instances n'ont pas de type déclaré. Ces classes ont été annotées par : *AuthorList*, *City*, *PublicationPage*, *HomePage*.

La table 6.4 montre les profils des types du jeu de données *BNF*. Nous pouvons voir à travers les probabilités des propriétés que les instances d'un type ne sont pas décrites par toutes les propriétés du type, par exemple seules 57% des instances de type *Work* sont décrites par la propriété *date*. De plus, la propriété *relator* est définie pour certaines instances de type *Manifestation*, mais elle n'est pas déclarée dans le schéma. Sa probabilité est de 0.8 et ses mesures de complétude et de conformité sont donc de 0.2 (1 - 0.8). Sa pertinence est cependant indéfinie car cette propriété n'est pas déclarée dans le schéma.

La table 6.5 montre les profils des types du jeu de données *DBpedia*. Nous présentons uniquement un aperçu des profils de ces types car le nombre de propriétés pour un type donné est très élevé, de l'ordre de 150 propriétés en moyenne. Nous pouvons voir à travers les probabilités des propriétés que les instances d'un type sont très hétérogènes : par exemple 26% des instances de type *Museum* sont décrite par la propriété $numberOfVisitors$.

TABLE 6.3 – Profils des types du jeu de données *Conference*.

Type	Profil
Organization	$\{(\overrightarrow{name}, 1), (\overrightarrow{member}, 1), (\overleftarrow{affiliation}, 1)\}$
InProceedings	$\{(\overrightarrow{maker}, 1), (\overleftarrow{made}, 1), (\overleftarrow{hasPart}, 1), (\overrightarrow{URL}, 1), (\overrightarrow{creator}, 1), (\overrightarrow{month}, 1), (\overrightarrow{title}, 1), (\overrightarrow{abstract}, 1), (\overrightarrow{year}, 1), (\overrightarrow{authorList}, 1), (\overrightarrow{isPartOf}, 1)\}$
Point	$\{(\overrightarrow{long}, 1), (\overrightarrow{lat}, 1), (\overleftarrow{basedNear}, 1)\}$
Presentation	$\{(\overrightarrow{isRoleAt}, 1), (\overrightarrow{heldBy}, 1), (\overleftarrow{hasRole}, 1), (\overrightarrow{holdsRole}, 1)\}$
TutorialEvent	$\{(\overrightarrow{dtend}, 1), (\overrightarrow{homepage}, 1), (\overrightarrow{isSubEventOf}, 1), (\overleftarrow{isRoleAt}, 1), (\overleftarrow{isSuperEventOf}, 1), (\overrightarrow{dtstart}, 1), (\overrightarrow{summary}, 1), (\overrightarrow{url}, 1), (\overrightarrow{hasRole}, 1)\}$
Untyped (AuthorList)	$\{(\overleftarrow{authorList}, 1)\}$
Person	$\{(\overleftarrow{heldBy}, 0.56), (\overleftarrow{member}, 0.52), (\overrightarrow{name}, 1), (\overrightarrow{affiliation}, 0.52), (\overrightarrow{holdsRole}, 0.56), (\overleftarrow{creator}, 0.54), (\overleftarrow{maker}, 0.53), (\overrightarrow{basedNear}, 0.37), (\overrightarrow{made}, 0.53)\}$
Untyped (City)	$\{(\overleftarrow{basedNear}, 1)\}$
Untyped (PublicationPage)	$\{(\overleftarrow{url}, 1)\}$
Untyped (HomePage)	$\{(\overleftarrow{homePage}, 1)\}$

La figure 6.6 montre la complétude, la pertinence et la conformité entre les types des jeux de données *Conference* (a), *BNF* (b) et *DBpedia* (c) et leurs schémas respectifs. Pour la source de données *Conference*, nous pouvons voir que les mesures de pertinence, complétude et conformité des types sont maximales. Cela est dû au fait que le schéma a été découvert à partir de la structure implicite des données, ce qui permet de refléter au mieux le contenu d'un jeu de données. Cependant, pour le type *Person*, ces mesures de qualité ne sont pas maximales car les instances de ce type sont hétérogènes. Pour améliorer la description du schéma, il faudrait introduire un schéma probabiliste pour les propriétés comme décrit dans la section 6.7.1. Pour le jeu de données *BNF*, les mesures sont variables d'un type à un autre. Les mesures de qualité ne sont pas maximales car il existe plusieurs propriétés utilisées pour décrire les instances et qui ne figurent pas dans le schéma, comme la propriété *relator*. Pour réduire l'écart entre les données et le schéma, il faudrait procéder à la découverte du schéma implicite des données

TABLE 6.4 – Profils des types du jeu de données *BNF*.

Type	Profil
Work	{($\overleftarrow{focus}$, 0.71), ($\overleftarrow{workManifested}$, 0.71), ($\overrightarrow{subject}$, 1), ($\overrightarrow{creator}$, 1), ($\overrightarrow{language}$, 0.85), ($\overrightarrow{title}$, 1), ($\overrightarrow{description}$, 0.85), ($\overrightarrow{dateOfWork}$, 0.71), ($\overrightarrow{date}$, 0.57)}
Concept	{($\overrightarrow{prefLabel}$, 1), ($\overrightarrow{editorialNote}$, 0.71), ($\overrightarrow{focus}$, 1), ($\overrightarrow{FRBNF}$, 1), ($\overrightarrow{identifier}$, 0.28)}
Person	{($\overleftarrow{creator}$, 1), ($\overleftarrow{contributor}$, 0.8), ($\overleftarrow{focus}$, 0.6), ($\overrightarrow{languageOfThePerson}$, 1), ($\overrightarrow{page}$, 1), ($\overrightarrow{placeOfDeath}$, 0.6), ($\overrightarrow{gender}$, 1), ($\overrightarrow{placeOfBirth}$, 0.8), ($\overleftarrow{auteur}$, 0.2),...}
Expression	{($\overleftarrow{expressionManifested}$, 0.5), ($\overrightarrow{contributor}$, 1)}
Manifestation	{($\overrightarrow{workManifested}$, 1), ($\overrightarrow{expressionManifested}$, 1), ($\overrightarrow{roles}$, 0.8), ($\overrightarrow{date}$, 1), ($\overrightarrow{publishersName}$, 1), ($\overrightarrow{relators}$, 0.8), ($\overrightarrow{dateOfPublication}$, 1), ($\overrightarrow{auteur}$, 0.8), ($\overrightarrow{identifiantRenouard}$, 1), ($\overrightarrow{sourceConsulted}$, 0.2), ($\overrightarrow{title}$, 1)}

TABLE 6.5 – Profils des types du jeu de données *DBpedia*.

Type	Profil
Country	{($\overrightarrow{name}$, 1), ($\overrightarrow{capital}$, 0.8), ($\overrightarrow{continent}$, 1), ($\overrightarrow{governmentType}$, 0.7), ($\overleftarrow{placeOfBirth}$, 0.33), ($\overleftarrow{deathPlace}$, 0.2), ($\overrightarrow{lat}$, 0.2), ($\overrightarrow{long}$, 0.2),...}
Politician	{($\overrightarrow{name}$, 1), ($\overleftarrow{primeMinister}$, 0.2), ($\overrightarrow{party}$, 0.73), ($\overrightarrow{children}$, 0.21), ($\overrightarrow{birthDate}$, 0.94), ($\overleftarrow{successor}$, 0.78), ($\overrightarrow{deathDate}$, 0.68),...}
Soccer Player	{($\overrightarrow{name}$, 1), ($\overrightarrow{team}$, 1), ($\overrightarrow{height}$, 0.46), ($\overrightarrow{birthDate}$, 1), ($\overleftarrow{currentMember}$, 0.8), ($\overrightarrow{placeOfBirth}$, 1), ($\overrightarrow{deathDate}$, 0.06),...}
Movie	{($\overrightarrow{writer}$, 1), ($\overrightarrow{genre}$, 0.88), ($\overleftarrow{created}$, 0.05), ($\overrightarrow{director}$, 1), ($\overrightarrow{abstract}$, 1), ($\overleftarrow{previousWork}$, 0.11), ($\overrightarrow{subTitle}$, 0.05), ($\overrightarrow{place}$, 0.05),...}
Book	{($\overrightarrow{author}$, 1), ($\overrightarrow{language}$, 0.71), ($\overrightarrow{genre}$, 1), ($\overleftarrow{lastAppearance}$, 0.07), ($\overrightarrow{title}$, 0.92), ($\overrightarrow{abstract}$, 1), ($\overleftarrow{previousWork}$, 0.57), ($\overleftarrow{successor}$, 0.07),...}
Museum	{($\overrightarrow{name}$, 1), ($\overrightarrow{depiction}$, 0.86), ($\overrightarrow{numberOfVisitors}$, 0.26), ($\overrightarrow{director}$, 0.26), ($\overleftarrow{city}$, 0.06), ($\overleftarrow{battle}$, 0.06), ($\overrightarrow{lat}$, 0.86), ($\overrightarrow{long}$, 0.86),...}

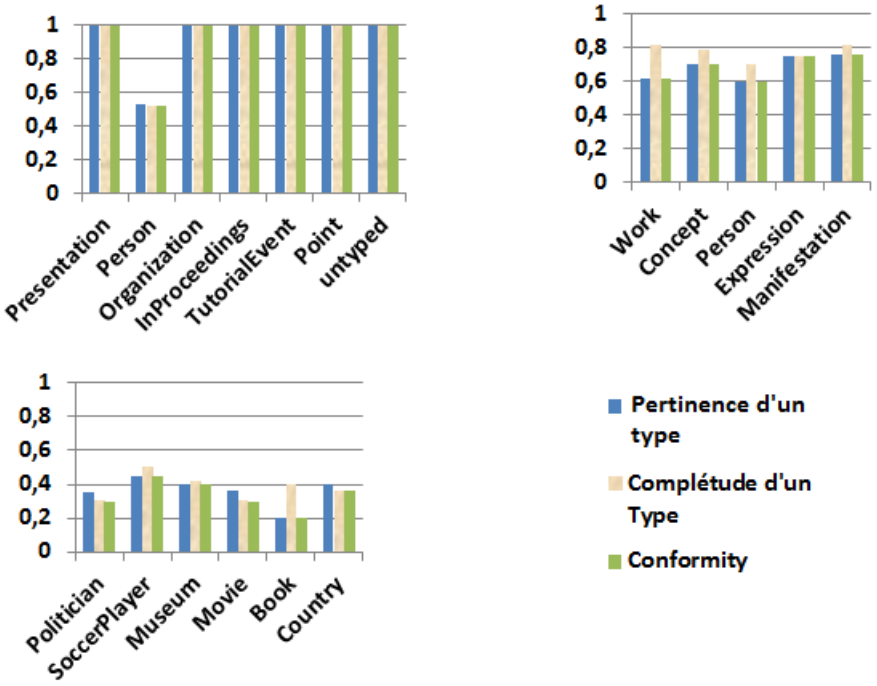


FIGURE 6.6 – Complétude des instances des types, pertinence des ensembles de propriétés des types et conformité des types des jeux de données *Conference* (a), *BNF* (b) et *DBpedia* (c).

qui est plus adapté pour décrire l'état des données dans leur version courante. Pour le jeu de données *DBpedia*, l'écart entre le schéma défini et le jeu de données est encore plus visible à travers des mesures de qualité ayant une valeur faible. En effet, comme vu dans les profils de types de ces données, les instances de même types sont très hétérogènes. Dans ce cas, le schéma utilisé ne permet pas de refléter le niveau d'hétérogénéité de ces données. L'utilisation d'un schéma probabiliste, comme proposé dans ce chapitre, serait plus approprié pour décrire ces données.

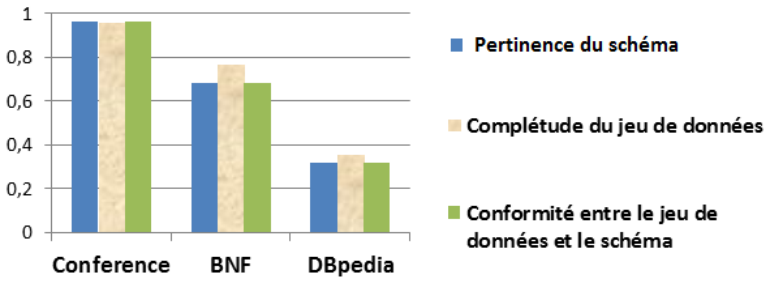


FIGURE 6.7 – Complétude d'un jeu de données, pertinence du schéma et la conformité des jeu de données *Conference* (a), *BNF* (b) et *DBpedia* (c).

La figure 6.7 montre les mesures de complétude, pertinence et conformité entre les jeux de données *Conference* (a), *BNF* (b) et *DBpedia* (c) avec leur schéma. Nous pouvons voir que le schéma découvert à partir de la structure implicite des données de *Conference* est plus conforme que le schéma explicite de *BNF* et de *DBpedia*. Le schéma de la *BNF* n'est pas pertinent vu que plusieurs propriétés utilisées pour décrire les instances n'ont pas été déclarées dans le schéma explicite. Il serait préférable dans ce cas de procéder à une découverte du schéma implicite du jeu de données. Le schéma du jeu de données extrait de *DBpedia* est très peu conforme car il ne peut pas refléter l'hétérogénéité très forte des données qu'il décrit. Il serait préférable dans ce cas de décrire les données avec un schéma probabiliste comme proposé précédemment.

6.10 Conclusion

Nous avons proposé une approche pour évaluer la conformité d'une source de données à son schéma. Nous avons également émis des propositions pour améliorer la qualité de la description d'un schéma de données liées afin de réduire l'écart entre une source de données et son schéma. À cette fin, nous avons construit un profil pour chaque type, constitué d'un ensemble de propriétés dans lequel chaque propriété est associée à une probabilité représentant la fréquence de cette propriété dans les instances du type. Nous avons également considéré l'ensemble de propriétés pour chaque type selon les déclarations dans le schéma. Afin d'évaluer la conformité entre une source de données et son schéma, nous avons proposé un ensemble de facteurs de qualité ainsi que les métriques associées. Nos expérimentations ont mis en évidence les disparités existantes entre les sources de données et leur schéma dans tous les jeux de données réels testés. Cette disparité est une conséquence de la flexibilité des langages utilisés pour exprimer les données liées. En effet, ces langages n'imposent aucune contrainte sur la structure des données, ce qui induit un écart entre le schéma déclaré et la réalité de la structure des données insérées.

La conformité entre un schéma et son jeu de données permet de renseigner sur l'écart entre un schéma et la source qu'il décrit. En effet, une valeur de conformité faible peut refléter le fait que les données ont beaucoup évolué par rapport au schéma déclaré initialement ; et cela peut inciter à faire une découverte du schéma implicite (voir chapitre 2) sur un jeu de données, qui sera plus à même de décrire l'état actuel des données. Une valeur de conformité faible, peut également indiquer que le jeu de données est très hétérogène, par le fait que des instances de même type sont décrites par des ensembles de propriétés différents. Ce qui est dû aux langages de description de schéma utilisés (RDF(S)/OWL) qui ne permettent pas d'exprimer l'hétérogénéité des données au sein du même type. Dans ce cas, l'extension du schéma probabiliste proposé dans la section 6.7.1 peut être utilisée. Celle-ci permet de décrire cette hétérogénéité en utilisant des probabilités, ce qui

permet de savoir à quel point une propriété déclarée dans le schéma est réellement renseignée dans les données, par exemple, afin d'évaluer la capacité d'une source de données à répondre à une requête. Dans le contexte d'une requête qui doit être exécutée sur des sources de données distribuées, un schéma probabiliste pour chaque source permettrait de guider la décomposition de la requête pour envoyer les sous-requêtes aux sources pertinentes. Les probabilités des propriétés pourraient être utiles pour optimiser le plan d'exécution en ordonnant les sous-requêtes en fonction de la sélectivité de leurs critères, mais aussi pour estimer les coûts d'un plan d'exécution.

Dans ce chapitre, nous nous sommes intéressés à l'évaluation de la complétude, de la pertinence et de la conformité entre une source de données et son schéma. Les métriques proposées ne sont pas adaptées pour évaluer la pertinence et complétude des super-types, ce qui peut constituer une voie à explorer pour des travaux futurs. Il serait également intéressant d'identifier d'autres facettes de la conformité pour les données du Web, qui exploitent d'autres déclarations concernant le schéma, comme *owl*:*InverseFunctionalProperty* qui définit une propriété comme ayant une valeur unique pour chaque élément du domaine, ou *owl*:*TransitiveProperty* qui définit une propriété comme transitive.

Chapitre 7

Conclusion générale

7.1 Bilan des contributions

L'émergence du Web sémantique a entraîné la publication d'un nombre croissant de sources de données décrites en RDF(S)/OWL, mais l'exploitation de ces sources de données souffre du manque d'informations décrivant leur contenu. Dans ce contexte, nous avons apporté des contributions à la résolution de ce problème.

Nous avons présenté dans le chapitre 3, une nouvelle approche qui permet la découverte du schéma d'une source à partir de la structure implicite de ses données, sans faire d'hypothèses sur l'existence de déclarations sur le schéma dans le jeu de données. Nous avons d'abord fait une étude sur les algorithmes de clustering pour le regroupement des données du Web sémantique. Nous avons montré que l'algorithme de clustering DBscan est celui qui répond le mieux aux caractéristiques de ces données, et ce car cet algorithme trouve des classes de forme arbitraire, il est robuste au bruit et déterministe ; de plus, le nombre de clusters n'est pas requis comme paramètre. Comme cet algorithme requiert de spécifier un seuil de similarité pour le clustering, nous avons proposé une approche qui permet de détecter automatiquement le seuil de similarité dans un jeu de données selon la distribution des plus proches voisins. Nous avons également adapté DBscan pour construire un profil probabiliste qui caractérise chaque type. Les profils des types servent à étendre DBscan et permettent d'attribuer plusieurs types à une même instance. Ils permettent également de trouver les types d'une nouvelle instance. Notre approche permet également de trouver les liens sémantiques et hiérarchiques entre les types. Les liens sémantiques sont découverts en analysant les profils de types, et les liens hiérarchiques sont découverts en appliquant un algorithme de clustering hiérarchique sur les profils de types, ce qui est beaucoup moins coûteux que de l'appliquer sur le jeu de données.

Le profil de type construit lors de la découverte du schéma implicite des données ne permet pas de renseigner la co-occurrence entre les propriétés. Nous avons proposé dans le chapitre 4, *SchemaDecrypt* qui est une approche permettant de découvrir en ligne les différentes versions structurelles d'un type sans avoir à parcourir ou à télécharger la source de données. La difficulté de cette tâche réside

essentiellement sur le fait que l'accès aux données se fait à travers des requêtes. De plus, il peut exister des restrictions d'accès appliquées par le serveur comme un timeout sur l'exécution d'une requête, ou la limitation du nombre de requêtes envoyées pour ne pas engorger le réseau. Pour résoudre le problème combinatoire auquel nous sommes confrontés lors de la génération des versions d'un type, nous avons construit un profil probabiliste de type pour : (i) guider l'exploration des versions candidates, (ii) tester les versions les plus probables d'abord, (iii) découvrir des règles d'occurrences entre les propriétés du type, et (iv) définir un critère d'arrêt. Nous réduisons l'espace de recherche des versions candidates grâce à des règles d'inclusion et d'exclusion. Celles-ci permettent également de savoir localement si une version peut avoir des instances sans envoyer la requête correspondante à la source, et aussi de réduire le nombre de propriétés dans une requête afin de minimiser son temps de traitement. Nous avons également proposé une codification binaire des versions candidates qui permet d'explorer les versions de manière ordonnée sans construire un graphe d'exploration. De plus, cette codification représente un gain important en mémoire. Nous avons également proposé *SchemaDecrypt++* dont l'idée est d'explorer certaines versions en parallèle afin d'optimiser le temps de traitement. La difficulté d'une exploration parallèle des versions réside dans le recouvrement potentiel pouvant exister entre les versions. Nous avons exploité les règles d'exclusions pour former des modèles de versions qui ne génèrent pas de versions candidates qui se recouvrent. Nous avons proposé de découvrir de nouvelles règles d'occurrences propres à chaque modèle de version, et des façons particulières de les exploiter pour réduire l'espace de recherche. Nous avons également proposé des élagages du graphe d'exploration parallèle. Cette approche requiert des déclarations sur le schéma pour pouvoir identifier les propriétés associées au type ainsi que ses instances, notamment les propriétés $rdfs{:}domain$, $rdfs{:}range$, $rdfs{:}subClassOf$ et $rdf{:}type$.

La découverte du schéma implicite d'une source de données permet de découvrir les types en regroupant les instances similaires structurellement. Elle fournit des groupes d'instances correspondant à des types. Nous avons proposé dans le chapitre 5 une approche d'annotation pour capturer la sémantique de ces groupes d'instances. Nous avons proposé quatre algorithmes d'annotation reposant sur des sources externes de connaissances disponibles sur le Web : un algorithme d'annotation utilisant le nom, un algorithme d'annotation utilisant les propriétés, un algorithme d'annotation utilisant le vocabulaire et l'annotation hybride qui combine les résultats des trois précédents algorithmes d'annotation. Chaque algorithme utilise les instances d'un type pour extraire ses annotations candidates. Nous avons également proposé une approche pour découvrir la hiérarchie de types dans une source de données à l'aide des annotations extraites. En revanche, les terminologies utilisées dans une source de données et dans des bases de connaissances peuvent être différentes. Nous avons proposé pour cela de considérer des formes canoniques et générer un ensemble de synonymes pour chaque propriété

en utilisant *WordNet* [88]. Ensuite, une recherche est effectuée dans la base de connaissances pour chaque synonyme jusqu'à ce que la propriété correspondante soit retrouvée.

Nous avons proposé dans le chapitre 6, une approche pour évaluer l'écart entre une source de données et son schéma. Nous avons proposé différents facteurs de qualité ainsi que les métriques associées. Les facteurs proposés sont la complétude de la source de données, la pertinence du schéma et la conformité entre le schéma et la source qu'il décrit. Nous avons également émis des propositions pour améliorer la qualité de la description du schéma d'une source exprimée en RDF afin de réduire cet écart, et cela, en proposant une extension du schéma qui permet de refléter la conformité à travers des probabilités. Nous avons également proposé une méthodologie qui permet d'adapter un schéma en fonction de l'évolution des données. Nos expérimentations ont mis en évidence les disparités existantes entre les sources de données et leur schéma dans tous les jeux de données réels testés.

7.2 Perspectives

Les travaux de recherche menés dans le cadre de cette thèse ouvrent de nombreuses perspectives. Dans cette section, nous allons présenter les différents axes à explorer dans des travaux futurs.

L'approche proposée de découverte du schéma implicite n'est pas adaptée au traitement d'une source de données massive. En effet, elle est fondée sur un regroupement par clustering qui nécessite une comparaison des instances deux à deux. Une perspective intéressante serait d'utiliser les technologies de *Big data* comme SPARK pour passer à l'échelle. L'approche de découverte des versions d'un type a montré que les instances d'un type suivent un certain nombre de patterns structurels. Par conséquent, il serait préférable d'appliquer d'abord une compression structurelle des données lors de la découverte du schéma implicite. En effet, une découverte de schéma implicite sur des instances donne exactement le même résultat qu'une découverte de schéma implicite sur des patterns exacts. De plus, les expérimentations faites dans [7] ont montré que par exemple pour la source de données DBpedia qui contient 29 688 668 instances, une compression structurelle réduit ce nombre à 1 333 098 patterns exacts.

L'approche de découverte de schéma proposée repose uniquement sur la structure implicite des données, il serait intéressant de pouvoir exploiter aussi les déclarations explicites sur le schéma quand elles sont fournies, et de proposer une approche hybride. La validation du schéma découvert représente également une tâche importante. Les déclarations sur le schéma fournies dans le jeu de données pourraient être utilisées pour cela : par exemple, deux entités ayant la même valeur pour la déclaration *rdf:type* doivent appartenir au même type découvert ; deux entités liées par une déclaration *owl:sameAs* doivent appartenir au même type découvert ; etc.

Le schéma découvert consiste en un ensemble de types et de liens entre eux. Cependant, d'autre informations peuvent le compléter comme : les propriétés symétriques, les types disjoints, les instances qui représentent le même objet, etc.

Dans notre approche, les types découverts ne sont pas remis en question lors de l'évolution des données. Cependant, il serait intéressant d'investiguer le cas d'une évolution importante des données qui pourrait modifier les types existants.

Le problème de définir les correspondances entre les propriétés de la source et les propriétés de la base de connaissances n'a pas été adressé lors de l'annotation des données. Il serait donc intéressant d'étudier comment l'annotation des types pourrait être utilisée pour résoudre l'hétérogénéité entre le schéma de différentes sources de données. Cela peut être envisagé par exemple en identifiant des classes ayant la même sémantique en comparant leurs ensembles respectifs d'annotations.

L'interrogation de sources de données distribuées est un problème important. Il serait intéressant d'investiguer l'utilisation de profils et les versions des types pour la décomposition des requêtes qui s'exécutent sur des sources de données distribuées et distantes. En effet, les versions peuvent contribuer à décomposer et à formuler les sous-requêtes pour obtenir la réponse la plus complète possible, et à optimiser le nombre de sous-requêtes envoyées. Le nombre d'occurrences de chaque version et les probabilités des propriétés fournies par les profils permettent d'optimiser et d'estimer le coût d'un plan d'exécution.

Dans notre travail, nous avons proposé une découverte des versions d'un type en ligne, sans avoir à télécharger la source pour parcourir ses données. Nous avons également proposé une approche de découverte de schéma pour une source de données locale. L'approche de découverte de schéma proposée repose sur un algorithme de clustering qui nécessite l'accès aux données afin de les comparer entre elles. Il serait intéressant d'étendre l'approche de découverte des versions d'un type en ligne pour permettre la découverte du schéma de la source de données en ligne.

L'interrogation d'une source de données en ligne doit tenir compte des restrictions d'accès imposées par le serveur de la source de données distante, comme par exemple une limite sur la taille maximale des résultats, un temps limité pour le traitement d'une requête ou encore un nombre limité de requêtes. Ces restrictions sont en place pour assurer que tout le monde à la même chance d'interroger les données du serveur et également pour se protéger des requêtes mal écrites et des robots. Ces restrictions sont d'autant plus strictes lorsque la source de données est très utilisée. Il serait intéressant de proposer un index basé sur le schéma implicite découvert afin de permettre la distribution de la source et ainsi d'alléger les restrictions d'accès aux données.

Nous avons proposé un ensemble de facteurs de qualité pour évaluer l'écart entre une source de données et le schéma qui la décrit. D'autres facteurs de qualité pourrait être proposés pour refléter la co-occurrence entre les propriétés, ainsi que la conformité par rapport à d'autres déclarations sur le schéma comme :

rdfs:*subClassOf*, *owl*:*InverseFunctionalProperty*, *owl*:*TransitiveProperty*, etc.

L'ensemble des contributions du présent travail est une première étape vers l'acquisition d'une connaissance fine du contenu d'une source de données, que les perspectives décrites ci-dessus permettraient d'enrichir.

Liste des publications

- **[SSDBM2017]** K. Kellou-Menouer and Z. Kedad. On-line Versioned Schema Inference for Large Semantic Web Data Sources. 29^{th} International Conference on Scientific and Statistical Database Management, SSDBM, 2017, ACM 12 pages.

- **[TLDKS2016]** K. Kellou-Menouer and Z. Kedad. A Self-Adaptive and Incremental Approach for Data Profiling in the Semantic Web. International Journal for Transactions on Large-Scale Data and Knowledge-Centered Systems, TLDKS, 108-133, 2016.

- **[ER2015]** K. Kellou-Menouer and Z. Kedad. Schema Discovery in RDF Data Sources. 34^{th} International Conference on Conceptual Modeling, ER, 481-495, 2015.

- **[OTM2016]** K. Kellou-Menouer and Z. Kedad. Class Annotation Using Linked Open Data. On the Move to Meaningful Internet Systems : OTM Conferences : CoopIS, C&TC, and ODBASE 2016, 709-726, 2016.

- **[ESWC2015]** K. Kellou-Menouer and Z. Kedad. Discovering Types in RDF Datasets. The Semantic Web : ESWC, 77-81, 2015. (Poster)

- **[QMMQ2015]** K. Kellou-Menouer and Z. Kedad. Evaluating the Gap between a Dataset and Its Schema. Workshop Quality of Models and Models of Quality in conjunction with the 34^{th} International Conference on Conceptual Modeling (ER), QMMQ, 283-292, 2015.

- **[EGC2015]** K. Kellou-Menouer and Z. Kedad. A Clustering Based Approach for Type discovery in RDF Data Sources. Extraction et gestion des connaissances, EGC, 471-472, 2015. (Poster)

- **[FDC2015]** K. Kellou-Menouer and Z. Kedad. Using Clustering for Type Discovery. Workshop Fouille de Données Complexes, FDC, 2015.

Bibliographie

[1] Klitos Christodoulou, Norman W. Paton et Alvaro A. A. Fernandes : Structure inference for linked data sources using clustering. *In Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, pages 60–67, 2013.

[2] Jesse Xi Chen et Marek Z Reformat : Learning categories from linked open data. *In International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 396–405. Springer, 2014.

[3] Svetlozer Nestorov, Serge Abiteboul et Rajeev Motwani : Inferring structure in semistructured data. *ACM SIGMOD Record*, pages 39–43, 1997.

[4] Qiu Yue Wang, Jeffrey Xu Yu et Kam-Fai Wong : Approximate graph schema extraction for semi-structured data. *In Advances in Database Technology EDBT 2000*, pages 302–316. Springer, 2000.

[5] Alexandre Delteil, Catherine Faron-Zucker et Rose Dieng : Learning ontologies from RDF annotations. *In Workshop on Ontology Learning*, 2001.

[6] Diego Sevilla Ruiz, Severino Feliciano Morales et Jesús García Molina : Inferring versioned schemas from NoSQL databases and its applications. *In International Conference on Conceptual Modeling*, pages 467–480. Springer, 2015.

[7] Fethi Belghaouti, Amel Bouzeghoub, Zakia Kazi-Aoul et Raja Chiky : Fregrapad : Frequent rdf graph patterns detection for semantic data streams. *In Research Challenges in Information Science (RCIS), 2016 IEEE Tenth International Conference on*, pages 1–9. IEEE, 2016.

[8] Mathias Konrath, Thomas Gottron, Steffen Staab et Ansgar Scherp : Schemex : efficient construction of a data catalogue by stream-based indexing of linked data. *Web Semantics : Science, Services and Agents on the World Wide Web*, 16:52–58, 2012.

[9] Mussab Zneika, Claudio Lucchese, Dan Vodislav et Dimitris Kotzinos : Rdf graph summarization based on approximate patterns. *In International Workshop on Information Search, Integration, and Personalization*, pages 69–87. Springer, 2015.

[10] Šejla Čebirić, François Goasdoué et Ioana Manolescu : Query-oriented summarization of rdf graphs. *Proceedings of the VLDB Endowment*, 8(12): 2012–2015, 2015.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander et Xiaowei Xu : A density-based algorithm for discovering clusters in large spatial databases with noise. *In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231, 1996.

[12] Linked Open Data Cloud (LOD Cloud), diagram. http ://lod-cloud.net/.

[13] Linked Open Vocabularies (LOV). http ://lov.okfn.org/dataset/lov/.

[14] The World Wide Web Consortium (w3c). `https://www.w3.org/`.

[15] Resource description framework - RDF. `https://www.w3.org/RDF/`,.

[16] Ressource description framework schema - RDFS. `https://www.w3.org/TR/rdf-schema`,.

[17] Web ontology language - OWL. `https://www.w3.org/OWL/`.

[18] Tom Heath et Christian Bizer : Linked data : Evolving the web into a global data space. *Synthesis lectures on the semantic web : theory and technology*, 1(1):1–136, 2011.

[19] Pierre-Yves Vandenbussche, Ghislain A Atemezing, María Poveda-Villalón et Bernard Vatant : Linked open vocabularies (lov) : a gateway to reusable semantic vocabularies on the web. *Semantic Web*, 8(3):437–452, 2017.

[20] Linked Open Data Cloud (LOD Cloud) cache, sparql endpoint. http ://lod.openlinksw.com/.

[21] Bastian Quilitz et Ulf Leser : Querying distributed RDF data sources with SPARQL. *In The Semantic Web : Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, pages 524–538, 2008.

[22] Serge Abiteboul, Marcelo Arenas, Pablo Barceló, Meghyn Bienvenu, Diego Calvanese, Claire David, Richard Hull, Eyke Hüllermeier, Benny Kimelfeld, Leonid Libkin, Wim Martens, Tova Milo, Filip Murlak, Frank Neven, Magdalena Ortiz, Thomas Schwentick, Julia Stoyanovich, Jianwen Su, Dan Suciu, Victor Vianu et Ke Yi : Research directions for principles of data management (dagstuhl perspectives workshop 16151). *CoRR*, abs/1701.09007, 2017.

[23] Ziawasch Abedjan, Toni Gruetze, Anja Jentzsch et Felix Naumann : Profiling and mining RDF data with ProLOD++. *In Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 1198–1201. IEEE, 2014.

[24] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener et Felix Naumann : Data profiling with metanome. *PVLDB*, 8(12):1860–1863, 2015.

[25] Yannis Papakonstantinou, Hector Garcia-Molina et Jennifer Widom : Object exchange across heterogeneous information sources. *In Data Engineering, Proceedings of the Eleventh International Conference on*, pages 251–260. IEEE, 1995.

[26] Klitos Christodoulou, Norman W Paton et Alvaro AA Fernandes : Structure inference for linked data sources using clustering. *In Transactions on Large-Scale Data-and Knowledge-Centered Systems XIX*, pages 1–25. Springer, 2015.

[27] Parisa D Hossein Zadeh et Marek Z Reformat : Context-aware similarity assessment within semantic space formed in linked data. *Journal of Ambient Intelligence and Humanized Computing*, 4(4):515–532, 2013.

[28] Markus Kirchberg, Erwin Leonardi, Yu Shyang Tan, Sebastian Link, Ryan KL Ko et Bu Sung Lee : Formal concept discovery in semantic Web data. *In Formal Concept Analysis*, pages 164–179. Springer, 2012.

[29] U PRISS : Formal concept analysis in information science. *Annual Review of In-formation Science and Technology*, 40:521–543, 2006.

[30] Dominik Brosius et Steffen Staab : Linked data querying through fca-based schema indexing. *In Proceedings of the 5th International Workshop "What can FCA do for Artificial Intelligence"? co-located with the European Conference on Artificial Intelligence, FCA4AI@ECAI 2016, The Hague, the Netherlands, August 30, 2016.*, pages 63–68, 2016.

[31] Svetlozar Nestorov, Serge Abiteboul et Rajeev Motwani : Extracting schema from semistructured data. *In ACM SIGMOD Record*, volume 27, pages 295–306. ACM, 1998.

[32] Roy Goldman et Jennifer Widom : Dataguides : Enabling query formulation and optimization in semistructured databases. 1997.

[33] Douglas H Fisher : Knowledge acquisition via incremental conceptual clustering. *Machine learning*, 2(2):139–172, 1987.

[34] Alexander Schätzle, Antony Neu, Georg Lausen et Martin Przyjaciel-Zablocki : Large-scale bisimulation of RDF graphs. *In Proceedings of the Fifth Workshop on Semantic Web Information Management*, page 1. ACM, 2013.

[35] Robin Milner : *Communication and concurrency*, volume 84. 1989.

[36] Johanna Völker et Mathias Niepert : Statistical schema induction. *In The Semantic Web : Research and Applications*, pages 124–138. Springer, 2011.

[37] R. Agrawal et R. Srikant : Fast algorithms for mining association rules in large databases. *In Proceedings of the 20th international conference on Very Large Data Bases (VLDB'94)*, pages 478–499. Morgan Kaufmann, September 1994.

[38] Heiko Paulheim : Browsing linked open data with auto complete. *Semantic Web Challenge*, 2012.

[39] Zijian Zheng et Geoffrey I Webb : Lazy learning of bayesian rules. *Machine Learning*, 41(1):53–84, 2000.

[40] Heiko Paulheim et Christian Bizer : Type inference on noisy RDF data. *In The Semantic Web–ISWC 2013*, pages 510–525. Springer, 2013.

[41] Andrea Giovanni Nuzzolese, Aldo Gangemi, Valentina Presutti et Paolo Ciancarini : Type inference through the analysis of wikipedia links. *In WWW2012 Workshop on Linked Data on the Web, Lyon, France, 16 April, 2012*, 2012.

[42] Jerome H Friedman, Forest Baskett et Leonard J Shustek : An algorithm for finding nearest neighbors. *IEEE Transactions on computers*, 100(10):1000–1006, 1975.

[43] Nansu Zong, Dong-Hyuk Im, Sung-Kwon Yang, Hyun Namgoong et Hong-Gee Kim : Dynamic generation of concepts hierarchies for knowledge discovering in bio-medical linked data sets. *In The 6th International Conference on Ubiquitous Information Management and Communication, ICUIMC '12, Kuala Lumpur, Malaysia, February 20-22, 2012*, pages 12 :1–12 :5, 2012.

[44] Lu Fang, Qingliang Miao et Yao Meng : Dbpedia entity type inference using categories. *In Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016.*, 2016.

[45] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli et Carlo Sartiani : Schema inference for massive JSON datasets. *In Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 222–233, 2017.

[46] Mussab Zneika, Claudio Lucchese, Dan Vodislav et Dimitris Kotzinos : Summarizing linked data RDF graphs using approximate graph pattern mining. *In Proceedings of the 19th International Conference on*

Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016., pages 684–685, 2016.

[47] Claudio Lucchese, Salvatore Orlando et Raffaele Perego : Mining top-k patterns from binary datasets in presence of noise. *In Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 165–176. SIAM, 2010.

[48] Petros Venetis, Alon Halevy, Jayant Madhavan, Marius Paşca, Warren Shen, Fei Wu, Gengxin Miao et Chung Wu : Recovering semantics of tables on the web. *Proceedings of the VLDB Endowment*, 4(9):528–538, 2011.

[49] Girija Limaye, Sunita Sarawagi et Soumen Chakrabarti : Annotating and searching web tables using entities, types and relationships. *Proceedings of the VLDB Endowment*, 3(1-2):1338–1347, 2010.

[50] Fabian M Suchanek, Gjergji Kasneci et Gerhard Weikum : Yago : a core of semantic knowledge. *In Proceedings of the 16th international conference on World Wide Web*, 2007.

[51] Gaëlle Hignette, Patrice Buche, Juliette Dibie-Barthélemy et Ollivier Haemmerlé : Fuzzy annotation of web data tables driven by a domain ontology. *In The Semantic Web : Research and Applications*, pages 638–653. Springer, 2009.

[52] Gianluca Quercini et Chantal Reynaud : Entity discovery and annotation in tables. *In Proceedings of the 16th International Conference on Extending Database Technology*, pages 693–704. ACM, 2013.

[53] Hector Gonzalez, Alon Y Halevy, Christian S Jensen, Anno Langen, Jayant Madhavan, Rebecca Shapley, Warren Shen et Jonathan Goldberg-Kidon : Google fusion tables : web-centered data management and collaboration. *In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1061–1066. ACM, 2010.

[54] Matthias Hagen, Maximilian Michel et Benno Stein : What was the query ? generating queries for document sets with applications in cluster labeling. *In Proceedings of the International Conference on Applications of Natural Language to Information Systems*, pages 124–133. Springer, 2015.

[55] Benno Stein et Sven Meyer Zu Eissen : Topic identification : Framework and application. *In Proceedings of the International Conference on Knowledge Management*, 2004.

[56] Bent Fuglede et Flemming Topsøe : Jensen-shannon divergence and hilbert space embedding. *In Proceedings of the International Symposium on Information Theory, ISIT*, page 31. IEEE, 2004.

[57] David Carmel, Haggai Roitman et Naama Zwerdling : Enhancing cluster labeling using wikipedia. *In Proceedings of the 32nd internatio-*

nal ACM SIGIR conference on Research and development in information retrieval, pages 139–146. ACM, 2009.

[58] Pucktada Treeratpituk et Jamie Callan : Automatically labeling hierarchical clusters. *In Proceedings of the International Conference on Digital Government Research*, 2006.

[59] David Milne et Ian H Witten : Learning to link with wikipedia. *In Proceedings of the 17th ACM conference on Information and knowledge management*, pages 509–518. ACM, 2008.

[60] Pablo N Mendes, Max Jakob, Andrés García-Silva et Christian Bizer : Dbpedia spotlight : shedding light on the web of documents. *In Proceedings of the 7th International Conference on Semantic Systems*, pages 1–8. ACM, 2011.

[61] Paolo Ferragina et Ugo Scaiella : Fast and accurate annotation of short texts with wikipedia pages. *IEEE Software*, 1(29):70–75, 2012.

[62] Christian Bizer, Tom Heath et Tim Berners-Lee : Linked data-the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[63] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak et Zachary Ives : DBpedia : A nucleus for a web of open data the semantic web. *The Semantic Web*, 2007.

[64] Jiawei Han et Micheline Kamber : Data mining : Concepts and techniques. 2006.

[65] Ming-Syan Chen, Jiawei Han et Philip S. Yu : Data mining : an overview from a database perspective. *IEEE Transactions on Knowledge and data Engineering*, 8(6):866–883, 1996.

[66] Nicola Fanizzi, Claudia d'Amato et Floriana Esposito : Metric-based stochastic conceptual clustering for ontologies. *Information Systems*, 34(8): 792–806, 2009.

[67] Anil K Jain, M Narasimha Murty et Patrick J Flynn : Data clustering : a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[68] Maria Halkidi, Yannis Batistakis et Michalis Vazirgiannis : On clustering validation techniques. *Journal of intelligent information systems*, 17(2-3):107–145, 2001.

[69] Anil K Jain : Data clustering : 50 years beyond k-means. *Pattern Recognition Letters*, pages 651–666, 2010.

[70] L. Kaufman et P. Rousseeuw : *Clustering by Means of Medoids*. Reports of the Faculty of Mathematics and Informatics. Faculty of Mathematics and Informatics, 1987.

[71] Pavel Berkhin : A survey of clustering data mining techniques. *In Grouping multidimensional data*, pages 25–71. Springer, 2006.

[72] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel et Jörg Sander : Optics : ordering points to identify the clustering structure. *In ACM Sigmod Record*, volume 28, pages 49–60. ACM, 1999.

[73] Alexander Hinneburg et Hans-Henning Gabriel : Denclue 2.0 : Fast clustering based on kernel density estimation. *In International symposium on intelligent data analysis*, pages 70–80. Springer, 2007.

[74] Wei Wang, Jiong Yang, Richard Muntz *et al.* : Sting : A statistical information grid approach to spatial data mining. 1997.

[75] Gholamhosein Sheikholeslami, Surojit Chatterjee et Aidong Zhang : Wavecluster : A multi-resolution clustering approach for very large spatial databases. *In VLDB*, volume 98, pages 428–439, 1998.

[76] James C Bezdek, Robert Ehrlich et William Full : FCM : The fuzzy C-Means clustering algorithm. *Computers & Geosciences*, 10:191–203, 1984.

[77] Arthur P Dempster, Nan M Laird et Donald B Rubin : Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, pages 1–38, 1977.

[78] David E. Goldberg : *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st édition, 1989.

[79] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider et Bernhard Seeger : The R*-tree : an efficient and robust access method for points and rectangles. *In ACM SIGMOD Record*, volume 19, pages 322–331. Acm, 1990.

[80] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer et Xiaowei Xu : Incremental clustering for mining in a data warehousing environment. *In Proceedings of the 24th International Conference on Very Large Data Bases*, pages 323–333. Morgan Kaufmann Publishers Inc., 1998.

[81] Belur V Dasarathy : Nearest neighbor (NN) norms : NN pattern classification techniques. 1991.

[82] C. Swenson : *Modern Cryptanalysis : Techniques for Advanced Code Breaking*. Wiley, 2012.

[83] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer *et al.* : DBpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[84] Linked Open Vocabularies (LOV), sparql endpoint. http ://lov.okfn.org/dataset/lov/sparql.

[85] Datalift project :. http ://datalift.org.

[86] Ondrej Zamazal et Vojtech Svátek : Oosp : Ontological benchmarks made on the fly. 2015.

[87] Philipp Frischmuth, Michael Martin, Sebastian Tramp, Thomas Riechert et Sören Auer : Ontowiki–an authoring, publication and visualization interface for the data web. *Semantic Web*, 6(3):215–240, 2015.

[88] Peter Oram : Wordnet : An electronic lexical database. christiane fellbaum (ed.). cambridge, ma : Mit press, 1998. pp. 423., 2001.

[89] Rakesh Agrawal, Ramakrishnan Srikant *et al.* : Fast algorithms for mining association rules. *In Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[90] Giuseppe Pirró : A semantic similarity metric combining features and intrinsic information content. *Data & Knowledge Engineering*, 68(11):1289–1308, 2009.

[91] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas et Octavian Udrea : Apples and oranges : a comparison of rdf benchmarks and real rdf datasets. *In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156. ACM, 2011.

[92] Carlo Batini et Monica Scannapieco : *Data Quality : Concepts, Methodologies and Techniques*. Springer Science & Business Media, 2006.

[93] T.C. Redman : *Data Quality for the Information Age*. Artech House, 1996.

[94] Richard Y Wang et Diane M Strong : Beyond accuracy : What data quality means to data consumers. *Journal of management information systems*, pages 5–33, 1996.

[95] Laure Berti-Équille, Isabelle Comyn-Wattiau, Mireille Cosquer, Zoubida Kedad, Sylvaine Nugier, Verónika Peralta, Samira Si-Said Cherfi et Virginie Thion-Goasdoué : Assessment and analysis of information quality : a multidimensional model and case studies. *International Journal of Information Quality*, 2(4):300–323, 2011.

[96] Daniel L Moody : Theoretical and practical issues in evaluating the quality of conceptual models : current state and future directions. *Data & Knowledge Engineering*, 55(3):243–276, 2005.

[97] Leo L Pipino, Yang W Lee et Richard Y Wang : Data quality assessment. *Communications of the ACM*, 45(4):211–218, 2002.

[98] Christian Fürber et Martin Hepp : Using semantic web resources for data quality management. *In Knowledge Engineering and Management by the Masses*, pages 211–225. Springer, 2010.

[99] Christian Fürber et Martin Hepp : Using sparql and spin for data quality management on the semantic web. *In Business Information Systems*, pages 35–46. Springer, 2010.

[100] Christian Fürber et Martin Hepp : Towards a vocabulary for data quality management in semantic web architectures. *In Proceedings of the 1st International Workshop on Linked Web Data Management*, pages 1–8. ACM, 2011.

[101] Christian Fürber et Martin Hepp : Swiqa - a semantic web information quality assessment framework. *In 19th European Conference on Information Systems, ECIS 2011, Helsinki, Finland, June 9-11, 2011*, page 76, 2011.

[102] Pablo N Mendes, Hannes Mühleisen et Christian Bizer : Sieve : linked data quality assessment and fusion. *In Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 116–123. ACM, 2012.

[103] Andreas Schultz, Andrea Matteini, Robert Isele, Pablo N Mendes, Christian Bizer et Christian Becker : Ldif-a framework for large-scale linked data integration. *In 21st International World Wide Web Conference (WWW 2012), Developers Track, Lyon, France*, 2012.

[104] Amrapali Zaveri, Dimitris Kontokostas, Mohamed A Sherif, Lorenz Bühmann, Mohamed Morsey, Sören Auer et Jens Lehmann : User-driven quality evaluation of dbpedia. *In Proceedings of the 9th International Conference on Semantic Systems*, pages 97–104. ACM, 2013.

[105] Dimitris Kontokostas, Amrapali Zaveri, Sören Auer et Jens Lehmann : Triplecheckmate : A tool for crowdsourcing the quality assessment of linked data. *In Knowledge Engineering and the Semantic Web*, pages 265–272. Springer, 2013.

[106] Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen et Amrapali Zaveri : Test-driven evaluation of linked data quality. *In Proceedings of the 23rd international conference on World Wide Web*, pages 747–758. ACM, 2014.

[107] Olaf Hartig et Jun Zhao : Using web data provenance for quality assessment. CEUR Workshop Proceedings, 2009.

[108] Olaf Hartig : Querying trust in rdf data with tsparql. *In The Semantic Web : Research and Applications*, pages 5–20. Springer, 2009.

[109] Heiko Paulheim et Christian Bizer : Improving the quality of linked data using statistical distributions. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(2):63–86, 2014.

[110] Marcelo Arenas, Gonzalo Díaz, Achille Fokoue, Anastasios Kementsietsidis et Kavitha Srinivas : A principled approach to bridging the gap between graph data and their schemas. *Proceedings of the VLDB Endowment*, 7(8):601–612, 2014.

[111] George A Miller : Wordnet : a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[112] Jun Zhao et Olaf Hartig : Towards interoperable provenance publication on the linked data web. *In WWW2012 Workshop on Linked Data on the Web, Lyon, France, 16 April, 2012*, 2012.