User-oriented exploration of semi-structured datasets

Nelly Barret

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2024IPPAX001

Thèse de doctorat

User-oriented exploration of semi-structured datasets

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Inria Saclay

École doctorale n°626 Ecole Doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique, Données et Intelligence Artificielle

Thèse présentée et soutenue à Palaiseau, le Vendredi 15 mars 2024, par

Mme, Nelly Barret

Composition du Jury :

Fatiha Saïs Professeur des universités, Université Paris Saclay, Laboratoire Interdisciplinaire des Sciences du Numérique (LISN)	Présidente
Jean-Marc Petit Professeur des universités, Insa Lyon, Laboratoire d'InfoRmatique en Image et Systèmes d'information (LIRIS)	Rapporteur
Olivier Teste Professeur des universités, Université Toulouse Jean Jaurès, Institut de Recherche en Informatique de Toulouse (IRIT)	Rapporteur (absent du jury)
Katja Hose Full professor, TU Wien, Databases and Artificial Intelligence Research Unit	Examinatrice
Stefano Ceri Professor, Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)	Examinateur
Fatemeh Nargesian Assistant professor, University of Rochester, Department of Computer Science	Examinatrice
Ioana Manolescu Chercheur senior, Inria et Ecole Polytechnique, Laboratoire d'Informatique de l'École Polytechnique (LIX)	Directrice de thèse
Karen Bastien Co-fondatrice, WeDoData	Co-encadrante de thèse

À mon compagnon, Julien

Et à ma famille proche :
ma maman Patricia et mon papa Michel
ma sœur jumelle Isabelle
et ma grand-mère Claudette

Pour leur amour et leur soutien sans faille

Nelly

Résumé

À travers le monde, la création, l'utilisation et le partage sans précédent des données contribuent à de nouvelles applications et opportunités économiques. Ces données sont souvent larges, hétérogènes en schéma comme en modèle, et plus ou moins structurées. Pour y mettre de l'ordre, le consortium du World Wide Web recommande de partager des graphes RDF, ce qui a été majoritairement adopté dans l'Open Data (*données ouvertes*), mais beaucoup d'autres formats sont utilisés en pratique. C'est le cas des journalistes qui récoltent des jeux de données de différents acteurs, qui ne se sont pas coordonnés. Par exemple, ils souhaitent comprendre comment les politiciens français en campagne sont liés, de près ou de loin, aux entreprises offshores. Dans ce cadre, ils ont trouvé sur la plateforme Kaggle des fichiers CSV recensant les pourcentages obtenus par les candidats aux élections présidentielles françaises ; les déclarations d'intérêt des parlementaires et sénateurs français sont partagées en XML sur le site du gouvernement ; les tweets des personnalités politiques françaises sont disponibles en JSON ; les graphes de propriétés (comme ceux de Neo4J) sont utilisés pour partager les données d'Offshore leaks, une base de données sur les sociétés offshores, dont certaines sont françaises. Les journalistes ont donc cruellement besoin d'outils pour consolider et interagir avec des sources provenant de différents acteurs, puis générer des résultats concrets qu'ils peuvent partager avec leurs collègues ou dans leur rédaction.

Plus généralement, les producteurs de données et les utilisateurs (novices ou non) qui doivent trouver, utiliser ou partager des jeux de données se trouvent face à un exercice difficile. Dans la littérature, il existe déjà ConnectionLens, un système capable de convertir des données structurées, semi-structurées et non-structurées en un graphe de données intégré. Ceci est très pratique pour intégrer et explorer plusieurs sources récoltées auprès de différents acteurs. En effet, le formalisme graphe permet une intégration fine et modulaire de l'information. En revanche, les utilisateurs qui ne sont pas familiers avec ce formalisme ou qui connaissent peu la structure et/ou le contenu de leurs jeux de données peuvent facilement se retrouver perdus.

Dans cette thèse, nous proposons deux nouvelles méthodes pour **appréhender, utiliser et partager des jeux de données semi-structurées**, i.e., documents XML, documents JSON, tableaux CSV, graphes RDF et de propriétés (PG). La première méthode produit des schémas rappelant les diagrammes Entité-Relation reflétant les entités et relations principales d'un jeu de données ; la seconde énumère un ensemble de chemins, considérés intéressants, entre des entités nommées, telles que des personnes, lieux, entreprises, etc. Ces deux propositions sont basées sur la vue graphe des différents modèles de données, introduite dans la littérature par ConnectionLens et ensuite étendue dans cette thèse sous la forme d'un graphe résumé (que nous appelons *graphe de collections*). Nous avons construit cette vue graphe et globale dans l'optique de tirer le meilleur de toutes ces données, quel que soit leur modèle.

La première partie de cette thèse se focalise sur la **génération automatique de descriptions de données**, sous la forme de diagrammes rappelant les schémas Entité-Relation, pour des données structurées et semi-structurées. Dans la littérature, il existe déjà beaucoup d'approches pour construire des schémas/descriptions pour les modèles semi-structurés, tels que JSON, XML, RDF et PG. En revanche, elles ne sont définies que pour un modèle à la fois, ce qui oblige à définir des méthodes de génération de schéma pour chaque modèle. Au contraire, l'approche proposée dans ce travail voit chaque jeu de données comme un ensemble d'entités (objets du monde réel), chacune ayant un ensemble de propriétés et pouvant avoir des connexions avec les autres. Par exemple, le jeu de données au format JSON décrivant les tweets de politiciens français contient deux entités (les tweets et les politiciens) et une relation (les tweets sont écrits par les politiciens).

La seconde partie de cette thèse se consacre à l'**énumération de chemins intéressants dans des données**. La plupart des travaux existants permettent aux utilisateurs de demander des mots-clés, tels que des noms de personnes ou d'entreprises, puis les systèmes retournent les données, ou chemins, associés à ces mots-clés. Cette approche est très pratique quand des mots-clés intéressants sont connus. Dans le cas contraire, et surtout si l'on connaît peu les données que l'on interroge, il est préférable d'adopter une approche plus générale, telle que celle que nous proposons. Ainsi, nous présentons une méthode qui, via quelques questions posées à l'utilisateur, énumère d'abord l'ensemble des chemins connectant des entités nommées, puis filtre et trie les chemins considérés comme les plus intéressants, et enfin matérialise ces chemins sur les données sous-jacentes. La qualité intéressante d'un chemin peut être calculée de différentes façons ; nous proposons de prendre en compte la fiabilité des entités extraites ainsi que la force de l'information présente dans le chemin.

Abstract

In the world, the unprecedented creation, use and share of data contribute to many new applications and economic opportunities. Such data is often large, heterogeneous in terms of schema and model, and more or less structured. To bring it order, the World Wide Web consortium recommends sharing RDF graphs, which has been mostly adopted in the Open Data initiative, but many other formats are used in practice. This is the case of journalists, who gather datasets from diverse actors, which did not cooperate. For instance, they seek to understand how French politicians in campaign relate to offshore companies. In that frame, they found on Kaggle CSV files describing the percentages obtained by candidates in the French presidential elections; French deputies' interest declarations are shared as XML on the governmental website; tweets from political figures are available in JSON; property graphs (such as pioneered by Neo4J) are used to share Offshore leaks data, a database about offshore companies, some of which are French. Therefore, journalists crucially need tools to consolidate and interact with sources gathered from various actors, then compute practical results they can share with their colleagues or within newsrooms.

More generally, data producers and users (newbies or not) who need to find, use or share datasets may have a hard time. In the literature, there already exists ConnectionLens, a system capable of converting structured, semi-structured and unstructured data in a integrated data graph. This is very convenient to integrate and explore sources from different actors. Indeed, the graph formalism allows a fine-granularity and modular integration of information. However, users who are not familiar with this formalism or who know very little about the structure and/or content of their datasets can quickly become lost.

In this thesis, we propose two novel methods to **grasp, use and share semi-structured datasets**, i.e., XML documents, JSON documents, CSV tables, RDF and Property Graphs (PG). The former method computes schemas akin to the classical Entity-Relationship diagrams, reflecting main entities of a dataset and their relationships; the latter enumerates a set of paths, considered interesting, between Named Entities, such as people names, places, companies names, etc. Those two proposals are based on the graph view of different data models, introduced in the literature by ConnectionLens, and further extended in this thesis in the form of a summarized graph (which we call *collection graph*). We built this global graph view in order to get the most out of all this data, regardless its model.

The first part of this thesis is about **automatically generating data descriptions**, in the form of diagrams like Entity-Relationship schemas, for structured and semi-structured data. There already exist many approaches to build schemas/descriptions out of semi-structured models, such as JSON, XML, RDF and PG. Nevertheless, they are defined for only one model at a time, requiring to define new methods for each model. On the contrary, our proposed approach sees each dataset as a set of

entities (objects in the real world), each of which having a set of properties and potentially some connections with other entities. For instance, the JSON dataset about tweets would encompass two entities (tweets and politicians) and a relationship (tweets are written by politicians).

The second part of this thesis focuses on the **enumeration of interesting paths in the data**. Most of the existing works allow users to ask keywords, such as people or companies names; then systems would return data or paths associated to those keywords. This approach is very suitable when users have some keywords of interest in mind. Otherwise, and even more if users know very little about the data they are querying, it is preferable to adopt a more general approach, such as the one we propose. After asking few questions to users, our method first enumerates the set of paths connecting Named Entities, then filters and sorts paths considered the most interesting, and finally materializes those paths on the underlying data. The interestingness of a path may be computed in several ways; we propose to take into account the reliability of Named Entities and the force of information present in the path.

Remerciements

Maintenant que j'ai défendu mon doctorat et que j'ai commencé ma nouvelle vie italienne, c'est le bon moment pour repenser à ces quatre dernières années. Quand j'ai commencé ma thèse en janvier 2021, j'ai rencontré l'équipe CEDAR à travers mon écran, mais des liens se sont très vite noués une fois les confinements terminés. Les présentations et conférences se sont ensuite enchaînées dans des cadres souvent idylliques. Sans crier garde, le moment de rédiger mon manuscrit est arrivé et c'était déjà bientôt la fin de mon doctorat.

Mes remerciements les plus chaleureux vont à Ioana, ma directrice de thèse, qui m'a encouragée à aller plus loin dans mes idées, à viser l'excellence, et à toujours acquérir de nouvelles compétences. Merci Ioana pour l'encadrement de qualité, la prévenance à mon égard et le soutien continu, malgré la pandémie de Covid-19 et ma vie lyonnaise. Je n'oublierai pas cette aventure, ni les précieux moments passés ensemble, que ce soit à CEDAR, dans Paris ou en conférence.

Je remercie ensuite les rapporteurs de ce manuscrit, Jean-Marc Petit et Olivier Teste, pour leur retours précis et constructifs. Merci aussi aux examinatrices et examinateurs, Fatiha Saïs, Katja Hose, Fatemeh Nargessian et Stefano Ceri : vous avez fait de ma soutenance un moment riche et gratifiant.

Merci aussi à Karen Bastien, data journaliste et co-fondatrice de WeDoData. Les cas d'utilisation et les données fournies nous ont permis de confronter notre production scientifique au monde réel. Data journaliste aussi, Camille a été de très bonne compagnie et toujours disponible pour des discussions passionnantes. Elles m'ont permis d'appréhender la pluridisciplinarité entre l'informatique et le journalisme, notamment par les échanges avec les journalistes, dont le groupe Datajournos.

Je n'oublie pas toute l'équipe CEDAR, avec qui j'ai eu des discussions épanouissantes au Magnan et lors de nos réunions d'équipe hebdomadaires. Plus particulièrement, je voudrais remercier Prajna pour notre riche collaboration pendant ma première année et Madhu pour avoir été une collègue et amie de grande qualité. Je n'oublie pas Théo G et Simon pour leur expertise ainsi que nos sorties au restaurant. Je pense aussi à Pawel, Yamen, Ghufran, Kun, Qi, Guillaume, Théo B, et tous les autres, avec qui j'ai partagé d'agréables moments et discussions.

Je remercie aussi les cinq stagiaires que j'ai co-supervisé pour leur investissement : Antoine, Jia Jean, Nikola, Tudor et Shay. Cette expérience a été très formatrice pour moi.

J'ai une pensée pour Fabien Duchateau et Franck Favetta, mes encadrants de stage en Licence et Master à Lyon. Merci de m'avoir permis d'entrer dans le monde de la recherche, ceci a conforté mon envie de faire une carrière académique. C'est avec vous que j'ai acquis les bases qui me sont utiles chaque jour.

Je souhaite remercier Inria Saclay, pour avoir été un endroit convivial pendant ces quatre dernières années, ainsi que la région Île-de-France et le projet SourcesSay pour avoir financé mon doctorat.

Mes derniers, mais pas des moindres, remerciements vont à mon compagnon et à ma famille. Julien, Maman, Papa, Isa, Mémé : merci de m'avoir soutenue et proposé des week-ends lyonnais, brulliolois, ou dompierrois pour décompresser.

Pour finir, je remercie toutes les personnes qui ont contribué, de près ou de loin, à cette aventure. J'ai vraiment apprécié d'avoir pu réaliser mon histoire doctorale à CEDAR. Merci à tous !

Contents

List of Figures

List of Tables

1

Introduction

Chapter Outline

Chapter Abstract. In this chapter, we first introduce the context of the thesis (Section 1.1), then, we present the main goals of the thesis (Section 1.2). Next, we list the main publications that came out of this work (Section 1.3) and the associated prototypes that have been implemented (Section 1.4). Finally, we outline the manuscript organization (Section 1.5).

1.1 Context

Data-driven applications are at the heart of many businesses and governmental initiatives. The domains of such applications are very varied, ranging from journalism to education, environment, or health. For instance, the RADAR application (from MediaCites, a French independent journal) monitors city mayors promises during their mandate, e.g., for the city of Lyon, 47 are completed, 70 are partially completed, 20 are uncompleted and 2 are unverifiable. Promises examples are: "increase the number of bus lines", "renovate 10K homes", "forbid advertising on buildings", etc. The evaluation of the promises success leverages city data. In the health domain, the COVID-19 pandemic allowed to massively share data and produce data-driven monitoring applications such as `https://covid19.who.int/data` and `https://coronavirus.jhu.edu/map.html`.

While a specific start date is hard to identify, we can generally refer to the so-called "Big Data era" as the last 20 years or so, more generally to a time when digital data has been recognized useful in a large variety of application settings, from science to entertainment, agriculture, across all industries and central to commerce [90]. Therefore, it becomes more and more important to design tools to store, manage, and understand data. Relational database systems [145] have been created in the late 60's to store and manage information. Such systems are suited to work with *traditional* data, in the sense of carefully gathered, prepared, stored, and shared data. Apart from database systems, files were used to share system configurations, results, and any other kind of reasonably small information. At that time, software and storage systems were designed based on an *application scenario*.

Following the rise of the Internet, the Big Data era shifted *traditional* data into *big* data. Many challenges have rapidly arisen from this new environment: it is not processable as before. The major difference with *big* data lies in the characteristics of the data, which can be described along the 7 following axes (also known as the 7V [100, 148]):

- **Volume.** First, the volume of data produced every day is enormous: according to the United Nations, 64.2 zettabytes of data has been created in 2020 (+314% compared to 2015). A large part of this is "data exhaust", i.e., passively collected data derived from everyday interactions with digital products or services, such as mobile phones, credit cards, and social media.
- **Velocity.** Next, data is shared as soon as it is produced, reinforcing the large spread of data, across all domains from health and market to environment and the arts. The 50 billion devices also play a role in data velocity, allowing, e.g., a sensor to directly send its data to a cloud or a student to order a pizza from her home. Striking examples of data velocity are: users watch 694,000 hours of YouTube videos, 16 million text messages are sent, 156 million emails are sent... every minute on the Web.
- **Variety.** Depending on many factors, such as the data producer preferences and the data application scenario constraints, dozens of different formats are used when sharing data: Web formats such as JSON and XML, open data formats such as RDF graphs, text such as Twitter and Facebook posts, data streams formats such as Avro, Parquet or ORC, etc. Data may also be photos, music, videos, etc. With the variety comes the heterogeneity: each data model has its own specificities and using data of different data formats requires dedicated algorithms. Further, all these data formats can be categorized in three main categories: structured, semi-structured and unstructured data (see Chapter 2).
- **Variability.** Produced data is not rigid, it may be updated at any time, e.g., to amend data

inconsistencies or to add new useful information. Variability means also more effort to obtain the same data quality.

- **Veracity.** With the large quantities of data produced every day (recall *volume* and *velocity*), it is very challenging to recognize and process erroneous data. The current most famous example is *fake news*. Also, there are 100 million spam emails sent every minute. Despite a start of global awareness about this, people still tend to share wrong, or at least unverified, information.

- **Visualization.** Visualization of data became critical in most domains at several levels. First, data producers need to understand their data in order to work with it. Also, citizens are more inclined to *learn* if they are provided graphical, vulgarized content. However, building visualization tools for big data is more challenging than ever.

- **Value.** Raw data is nowadays very cheap, mainly because of the global volume of data, but also because it is less expensive to store huge amounts of data. On the contrary, the real value resides in the cleaned, processed data as well as the insights and knowledge we can extract and build from it.

All these characteristics play a role in the increasing complexity of (big) data processing. This also stifles computer scientists and data experts to build data-driven applications. However, a large part of the daily-generated data is owned by (private) companies, thus limiting the global exchange and possibilities, due to company privacy restrictions. Nevertheless, in order to make data more accessible, the **Open Data initiative** emerged in the last decade. This aims at sharing datasets freely on the Internet and let the knowledge circulate through institutions and companies. Toward this goal, the World Wide Web Consortium recommends sharing data as **RDF** graphs, and this has been widely adopted, e.g., to build the Linked Open Data Cloud. However, practitioners also use a variety of other data formats such as relational data, XML or JSON documents and property graphs (recall *variety* and *heterogeneity*). Thousands of **CSV** datasets are available on Kaggle and the French public portal data.gouv.fr. **XML** is used to share bibliographic notices on PubMed, a leading website in the medical domain. **JSON** has become the reference model for the French parliament to increase the transparency of the public life, notably on the websites NosDeputes.fr and NosSenateurs.fr. **Relational databases** are sometimes shared as dumps, including schema constraints such as primary and foreign keys, or as CSV files. **Property graphs** (PGs, in short, such as pioneered by Neo4J) are used to share Offshore leaks, a journalistic database of offshore companies, or by the LDBC council.

More recently, a new community came up in the journalism community: the **data journalism**, also named **data-driven journalism** [36]. It aims at creating journalistic stories out of gathered, prepared, and filtered data, which requires a minimum of IT skills. Data journalism helps make a step toward exposing and explaining how society functions. One world-wide example of data journalism is the Panama Papers scandal. In 2016, more than 10 million confidential documents leaked and have been sent to the ICIJ (International Consortium of Investigative Journalists). It has disclosed confidential information about offshore companies used for tax evasion or money laundering. The ICIJ has cross-checked the information with several sources before a court case took place. This event helped data journalists to understand the need of IT tools to store and manage data, even if they are not highly skilled in computer science.

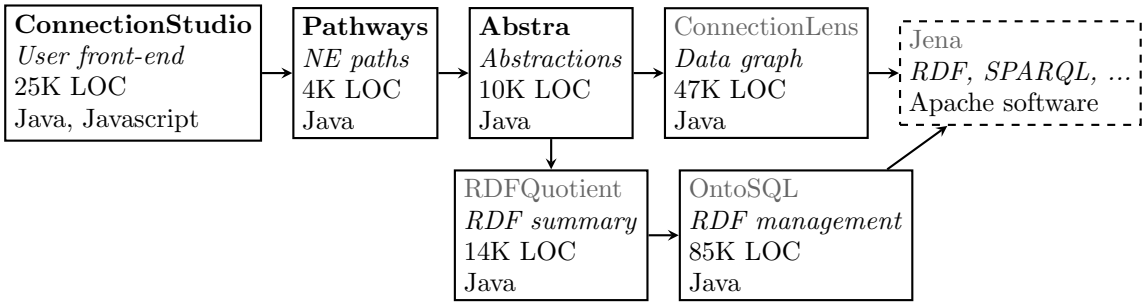


Figure 1.1: The software pile: from existing contributions (OntoSQL, RDFQuotient, ConnectionLens) to the original ones (Abstra, PathWays, ConnectionStudio) proposed in the thesis.

1.2 Goal of the thesis and main contributions

In the current data-driven world, it is clear that we need tools to manage and explore all of this data. However, data is produced in various formats (recall *variety*) and users need user-friendly interfaces to interact with the data (recall *visualization*), therefore the task of creating such tools becomes complex. In this thesis, our goal is to **propose tools to help users in the task of exploring heterogeneous data sources**. For this, we work on three different, but complementary, axes. The first one, presented in Section 1.2.1, aims at producing data abstractions in the form of lightweight Entity-Relationship diagrams. The second, described in Section 1.2.2, seeks to explore popular entities connections in the data. The last one, outlined in Section 1.2.3, intend to provide a complete user-friendly interface to load, query, clean and interact with heterogeneous data. Finally, we describe the associated scientific contributions in Section 1.2.4. All this work is based on ConnectionLens, a data lake that ingests (very) heterogeneous data and extracts, using language models, entities such as people names, companies names, places, emails, temporal references (data and time), etc. ConnectionLens also provides a keyword search feature, allowing users to explore the data using keywords of interest; the results they obtain are trees connecting one node that matches each query keyword. Research on keyword search algorithms is out of the scope of this thesis [13, 12, 44]. Figure 1.1 illustrates the software pile resulting from both the existing works (in grey) and the original ones (in bold) implemented in this thesis. For each system, we provide its name, its functionality (in italic), its number of lines of code (LOC) and the main programming languages used. An arrow from a software A to a software B signifies that A builds on B. Jena is outlined in a dashed box because it is an external software, provided by Apache.

1.2.1 Data abstractions for heterogeneous datasets with Abstra

When using data, e.g., to build a software, practitioners need to get a basic **understanding of a dataset content in order to decide whether it suits their needs**. This is a difficult task, especially for non-IT users such as data journalists, because this requires IT skills to understand the dataset technical features, e.g., read an XML document requires some basic knowledge about the markup language. Also, when the dataset is large, it is not always possible to understand it, even for IT experts. On the other side, data producers need tools to **generate automatically a description of the dataset** they want to share: this is essential to make it (re)usable and useful

for others.

Datasets descriptions generally take the form of a documentation or a schema. While these can help users in their quest of the right dataset, they have limitations. First, **documentation** is often lacking or insufficient (writing documentation is time-consuming). It may also be outdated, due to data evolution (recall *variability*). Next, a **schema** may be inferred from the dataset to describe its structure. However, schemas have several limitations:

1. They are rare when sharing semi-structured datasets such as XML, JSON, RDF and PG.
2. Schema *syntactic details*, such as regular expressions, are *hard to interpret* for non-IT users.
3. Generated schemas are often too large to be visualized as a whole (recall *visualization*).
4. Existing (schema inference) techniques[1] mainly focuses on the dataset structure, not on its *content*. It does not take advantage of linguistic information available in structures and text values.
5. They do not reflect *quantitatively* the dataset either, whereas showing only the most important structures in the dataset may give a sufficiently good overview of the data without overwhelming the user.

Data summarization techniques [79, 43, 48, 19, 157, 113], whose aim is to generate a schema of a semi-structured data model, partially lift limitation (1). In the particular case of RDF graphs, an *ontology* may accompany the graphs and give an overview of the semantic of the dataset, thus lifting limitation (4) but not the others. *Pattern mining* [87] may help users to grasp the popular patterns in their datasets, e.g., items often purchased together. This allows to bypass limitations (1) and (5) only. Finally, drawbacks (2) and (3) are generally left besides in schema generation works.

To answer practitioners' and data producers' needs, we present a **novel approach for abstracting any tabular, tree-structured, or graph-structured dataset**. Data abstraction takes the form of a lightweight Entity-Relationship schema, showing the most important entities in the data and how they relate to each other. Figure 1.2 shows the data abstraction produced on an XMark [150] XML document, describing an auction website where people sell and buy items through auctions. When a transaction is approved, it becomes a closed auction. Moreover, items are categorized, e.g., food, high-tech, home appliance, etc. People also may be interested in some item categories.

We focus on **application datasets**, each describing a specific scenario, e.g., the one in Figure 1.2, and do not consider "universal" datasets, such as Wikidata [161] and YAGO [138]. No universal dataset abstraction is likely to be *both* compact and comprehensive; extracting an application dataset from a universal one, based on keywords or terms the user already knows, is an orthogonal problem, e.g., [71, 46].

1.2.2 Entity-to-entity paths exploration with PathWays

In the data journalism context, when practitioners investigate data to work with, they generally look for interesting entities and connections (or *paths*) between them. Entities may have multiple forms; the most identifiable ones are the Named Entities (NEs in short), which may be very varied, e.g.,

[1] Large language models (LLM) recently demonstrated their capabilities in summarizing datasets into small texts. We prefer a more structured approach, as outlined in Section 3.4 and detailed in Chapter 6.

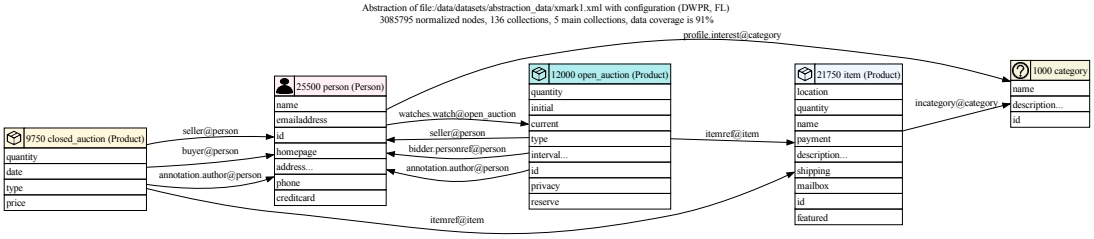


Figure 1.2: E-R model computed by Abstra from an XMark [150] XML document (3M nodes).

people, places, companies, dates, URIs, email addresses, etc. After being identified and extracted from the data, data journalists look for connections between them.

For instance, an investigative journalist works on conflicts of interest in the biomedical domain. More particularly, she works on bringing into light undeclared conflicts of interest, e.g., when scientific researchers get paid or influenced by external companies. For this investigation, she gathered some scientific publications (co-)authored by Helmut Greim, a German Professor of Toxicology, as PDF files and looks at the acknowledgment section. Concerning one paper discussing transparency in the biomedical domain, she finds out that authors, including H. Greim, declare no conflict of interest in the acknowledgement section ("[...] We claim no conflict of interest [...]"). In parallel, she has access to PubMed [193], a 36 million bibliographic notices database for biomedical literature. From there, she gathered, as XML documents, all the PubMed notices mentioning Helmut Greim. An example of a notice is presented in Figure 1.3. One of the PubMed notices represents a paper, co-authored by H. Greim, stating that the glyphosate molecule does not have particular negative effects on the human body. In that notice, she discovers in the acknowledgment section that authors received money from Monsanto, the company which created the RoundUp, a very efficient, but also very poisonous, glyphosate-based herbicide. This is one example among the numerous undeclared conflicts of interest in the biomedical domain.

From this example, we can derive the following set of observations:

1. Correspondences are hard to find, especially when it comes to join several datasets, potentially of different models.
2. Formulating queries over large, heterogeneous data is not feasible, especially in the frame of investigative journalism.
3. Entity-to-entity paths should not make any assumption on the edge directions, which strongly depends on how the data is modeled.
4. Entity-to-entity path enumeration is a very costly task, especially if the graph is large and/or there are many paths (the latter is almost always true, if the data is complex/heterogeneous, and/or if we allow paths to traverse edges in both directions).
5. Only *interesting and reliable* paths should be shown to users.

Observation (1) shows the importance of a multi-model approach, in order to connect different actors, appearing in different data sources. Observation (2) pushes forward the need of path enu-

```
<PubmedArticleSet>
  <PubmedArticle>
    <ArticleTitle>Evaluation of carcinogenic potential of the
    herbicide glyphosate[...]</ArticleTitle>
    <JournalTitle>Critical reviews in toxicology</JournalTitle>
    <pubmedLink>https://pubmed.ncbi.nlm.nih.gov/25716480/</pubmedLink>
    <Year>2015</Year>
    <DOI>10.3109/10408444.2014.1003423</DOI>
    <KeywordList>
      <Keywords>carcinogenicity</Keywords>
      <Keywords>Roundup</Keywords>
      ...
    </KeywordList>
    <AuthorList>
      <Author>
        <Name>Helmut Greim</Name>
        <Affiliation>Technical University Munich, Germany</Affiliation>
      </Author>
      ...
    </AuthorList>
    <CoiStatement>[...] Quality control and review of data transcription were
    valued services provided by Carrie Leigh Logan and Aparna Desai Nemali,
    Monsanto Quality Assurance Specialists.</CoiStatement>
  <PubmedArticle>
</PubmedArticleSet>
```

Figure 1.3: XML PubMed notice of the paper about glyphosate.

meration instead of user queries. Existing works allow to query heterogeneous data [160], if one can query the data. To illustrate observation (3), think of the following paths: *H. Greim* $\xrightarrow{authorOf}$ *PubMedPaper* $\xrightarrow{acknowledge}$ *Monsanto* and *H. Greim* $\xleftarrow{writtenBy}$ *PubMedPaper* $\xrightarrow{acknowledge}$ *Monsanto*; they are both interesting. Note that traversing a graph as if it is undirected considerably increases the possible entity paths, and thus the computational cost. For observation (5), non-expert users, or users which are not familiar with the dataset structure, cannot be expected to state "only the paths that they would like to see", since they lack technical expertise and/or dataset knowledge. However, *if prompted by the system*, they can give valuable input on whether *certain links (or connections) are worth making*, or whether they are just spurious links that would generate uninteresting paths. An example of uninteresting path is: in the PubMed example, all papers are written by authors, thus each paper is connected to a set of people. A more interesting example is shown in Figure 1.4: the user can explore how publications authors relate to company names found in their conflict-of-interest statement (COI statement). To answer above observations, users need **entity-to-entity paths to visualize connections between different actors, that may be shared across heterogeneous datasets**.

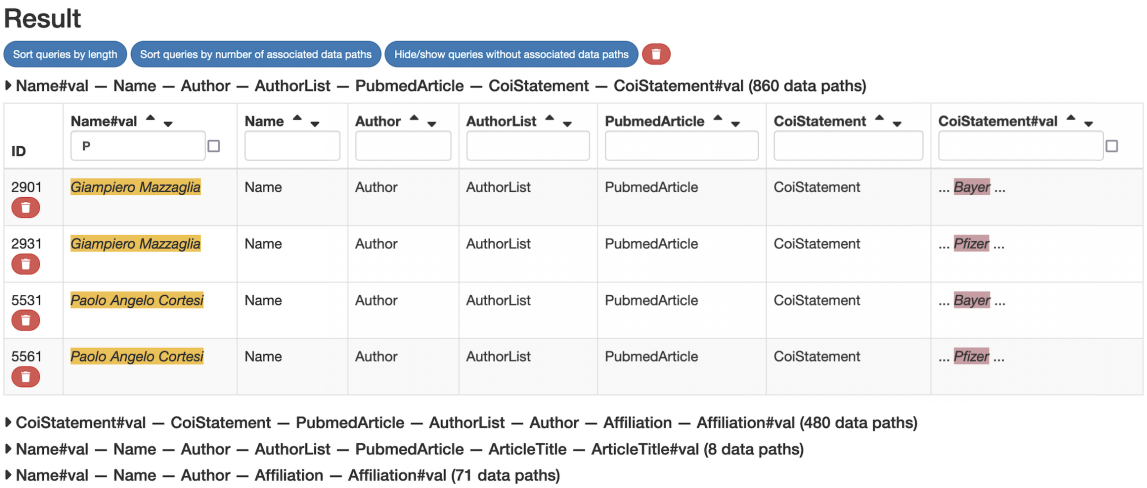


Figure 1.4: Tabular connections computed by PathWays from an XML PubMed bibliographic information (60K nodes).

1.2.3 Heterogeneous data exploration with ConnectionStudio

Developed in 2023, ConnectionStudio is a novel front-end to ConnectionLens, Abstra and PathWays; beyond integrating them, it provides some new functionalities, in particular helping users clean and query the data. For instance, Figure 1.5 shows a user query on the HATVP dataset, an open-source XML document provided by the French state to improve transparency in public life. Users express this query by combining and editing ("stitching") paths present in the data; the paths are enumerated by ConnectionStudio which proposes them in a drop-down menu. The data cleaning module offers ways to clean semi-automatically the data. For instance, in Figure 1.5, one can see that there is a missing hyphen in "alain pierre" (the name of the first deputy). By correcting this value, all identical values will be updated too, allowing a fast correction.

1.2.4 Scientific contributions

Our contributions rely on very heterogeneous data, ranging from structured (CSV, relational databases), to semi-structured (JSON, XML, RDF, Property Graphs) data. To tackle this heterogeneity, we transform incoming data as a data graph using ConnectionLens. Our contributions toward addressing these challenges are as follows:

1. First, we introduce a novel structure, called the *collection graph* to **summarize and represent efficiently large data graphs**. This structure is at the heart of the dataset abstraction and entity path exploration researches.

2. For dataset abstraction, we design an efficient algorithm to **select entities and relationships** constituting the final Entity-Relationship schema. This algorithm works on complex, and potentially cyclic, collection graphs.

3. For entity path exploration, we propose an efficient algorithm to enumerate entity paths connecting entities of interest to the user. This relies on a **view (sub-paths) recommender**

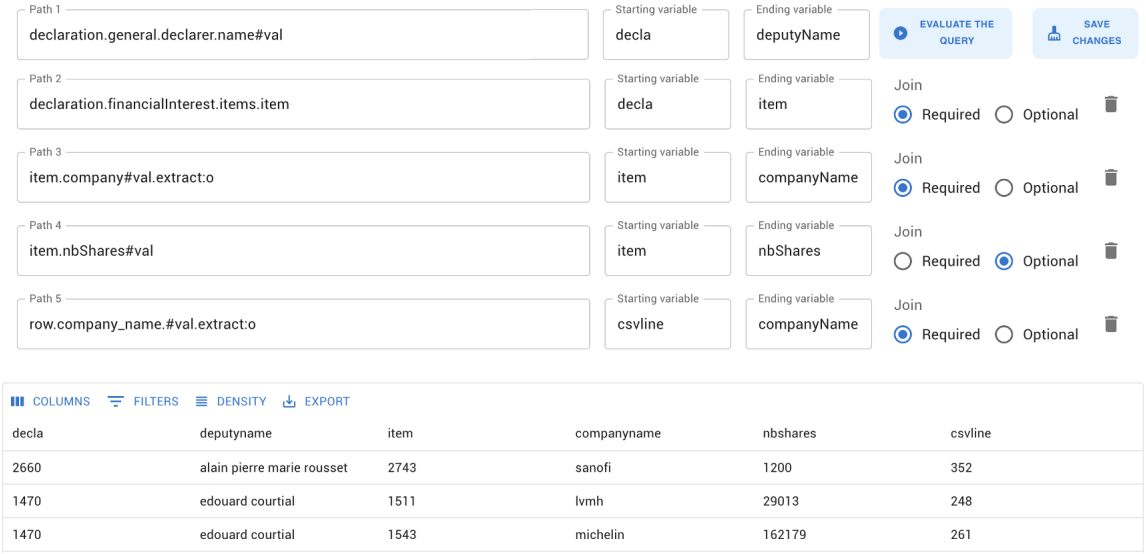


Figure 1.5: A query over the French assembly data showing how many shares each member has in the 40 most influential French companies.

system, which determines how to efficiently evaluate the set of entity paths, possibly overlapping.

4. For heterogeneous data exploration, we provide new ways to query the data, even if the dataset is unknown to the user. Specifically, we propose an interface to **incrementally build queries using elementary query building blocks**, computed over the integrated heterogeneous data. We also provide a module to **correct and improve the data graph** through semi-automatic processes.

1.3 Publications

The work presented in this thesis has been presented in national and international conferences. Specifically, we wrote five papers for Abstra, three for PathWays and three for Connection-Studio. The following list presents all the papers accepted for publication sorted by date (from the most to the least recent). Moreover, each international publication has also been informally presented at BDA, the annual French database conference.

- Abstra:

  1. Nelly Barret, Ioana Manolescu, Prajna Upadhyay. "Computing generic abstractions from application datasets". Researcher paper in *EDBT* 2024 (A).

  2. Nelly Barret, Ioana Manolescu, Madhulika Mohanty, Tudor Enache. "Finding the PG schema of any (semi)structured dataset: a tale of graphs and abstraction". Short paper in *ICDE SEAGraph workshop* 2024 (A*).

3. Nelly Barret, Ioana Manolescu, Prajna Upadhyay. "Abstra: toward generic abstractions for data of any model". Demonstration in *CIKM* 2022 (A).

4. Nelly Barret, Ioana Manolescu, Prajna Upadhyay. "Toward generic abstractions for data of any model". Short paper in *BDA* 2021 (national).

5. Nelly Barret. "Facilitating heterogeneous dataset understanding". PhD student paper in *BDA* 2021 (national).

- PathWays:

1. Nelly Barret, Antoine Gauquier, Jia Jean Law, Ioana Manolescu. "Finding meaningful paths in heterogeneous graphs with PathWays". Journal paper under review in *Information Systems* 2024.

2. Nelly Barret, Antoine Gauquier, Jia Jean Law, Ioana Manolescu. "Exploring heterogeneous data graphs through their entity paths". Research paper in *ADBIS* 2023 (C).

3. Nelly Barret, Antoine Gauquier, Jia Jean Law, Ioana Manolescu. "PathWays: entity-focused exploration of heterogeneous data graphs". Demonstration in *ESWC* 2023 (B).

- ConnectionStudio:

1. Oana Balalau, Nelly Barret, Simon Ebel, Théo Galizzi, Ioana Manolescu, Madhulika Mohanty. "Graph lenses over any data: the ConnectionLens experience". Short paper in *ICDE SEAGraph workshop* 2024 (A*).

2. Nelly Barret, Simon Ebel, Théo Galizzi, Ioana Manolescu, Madhulika Mohanty. "User-friendly exploration of highly heterogeneous data lakes". Demonstration in *EGC* 2024 (national).

3. Nelly Barret, Simon Ebel, Théo Galizzi, Ioana Manolescu, Madhulika Mohanty. "User-friendly exploration of highly heterogeneous data lakes". Short paper in *CoopIS* 2023 (B).

Five students have been involved in the work carried out in the thesis and are present in the above-mentioned publications. Antoine Gauquier and Jia Jean Law worked on PathWays, while Nikola Dobričic worked on ConnectionStudio. Shay Pripstein and Tudor Enache respectively worked on extending ConnectionLens and Abstra (see Chapter 8). I co-supervised all of them, along with Ioana Manolescu and Madhulika Mohanty.

1.4 Prototypes

Each project has its own website, sharing a quick introduction to the software, the code and a gallery of results:

- Abstra is available at: https://team.inria.fr/cedar/projects/abstra;
- PathWays is available at: https://team.inria.fr/cedar/projects/pathways;
- ConnectionStudio is available at: https://connectionstudio.inria.fr/.

1.5 Manuscript outline

The manuscript is organized as follows:

Chapter 2. We first introduce concepts that will be used in the thesis. We cover basic notations and existing structured and semi-structured data models.

Chapter 3. This work lies between heterogeneous data integration, data summarization, and structured and unstructured querying. Thus, we discuss existing works in those research areas and showcase their limitations.

Chapter 4. We explain how any semi-structured dataset can be viewed as a data graph. This is the key to provide solutions for well-known semi-structured data models.

Chapter 5. Building on the data graph, we introduce the collection graph, a core structure that is at the heart of data abstractions and entity-to-entity path enumeration. It summarizes both data structure and content, while being sufficiently small to allow designing efficient algorithms.

Chapter 6. Starting from the collection graph, we build data abstractions, i.e., schemas recalling Entity-Relationship diagrams and showing the main entities of a dataset and their relationships.

Chapter 7. Also based on the collection graph, we efficiently and interactively compute paths in the data graph connecting entities of interest, e.g., people, places, companies, etc. We also compute path interestingness, an important measure to rank meaningful paths first.

Chapter 8. We conclude this thesis manuscript and emphasize future works and open questions.

2

Preliminaries

Chapter Outline

Chapter Abstract. This chapter introduces key notions used throughout this manuscript. We first recall basic notions (Section 2.1). Second, we introduce the Entity-Relationship model (Section 2.2). Next, we describe the main data models we will consider: relational data (Section 2.3), XML documents (Section 2.4), JSON documents (Section 2.5), RDF graphs (Section 2.6) and property graphs (Section 2.7). More generally, data models can be grouped in three categories:

- **Structured data** conforms to a *schema*, designed beforehand and for a given application scenario. Structured information allows highly efficient algorithms, optimizations and data integrity but is not very flexible, e.g., newly upcoming data has to match the schema. Relational data is the leading data model when talking about structured data.
- **Unstructured data** models store data *as it is*, without any schema or kind of organization. Examples of unstructured data are text, PDF[1], binary files of any kind, etc.
- More recently, **semi-structured data models** appeared as a compromise between structured and unstructured data models. They organize data without relying on a fixed schema to conform with. On one hand, the adaptable organization better handles newly incoming data than a fixed schema. On the other hand, semi-structured data cannot benefit from all the inherent strengths of structured data. Therefore, many works [39, 199, 31, 35, 52] push toward bringing such strengths into semi-structured data models. Among the most widely used semi-structured data models, there are: XML, JSON, RDF and Property graphs.

[1]Tagged PDF have been created to transform PDF documents into semi-structured documents by annotating the document with HTML-like tags. This lifts only partially the problem because such annotation is time-consuming and of medium quality when performed automatically by a program.

2.1 Basics

We define the following domains:

- $\mathbb{N}$: the set of natural numbers, e.g., 0, 1, 2, 3, etc;
- $\mathbb{Z}$: the set of relative numbers, also known as *integer* values, e.g., 0, 1, -1, etc;
- $\mathbb{R}$: the set of real numbers, also known as *float* values, e.g., 1.34, -4.56, $\frac{2}{3}$, etc;
- $\mathbb{I}$: the set of identifier labels, e.g., myNode, person, city, etc;
- $\mathbb{S}$: the set of strings, e.g., "Paris", "Hello", "He is 12.", etc;
- $\mathbb{U}$: the set of URIs, e.g., `https://www.google.com/`, `https://dbpedia.org/page/Paris`, etc. They are often written in their concise form by simplifying the URI prefix, e.g., `dbpedia:Paris` (we will adopt this convention in this manuscript);
- $\mathbb{T}$: the set of time references, including date and time, e.g., 01/01/2021, 05:46PM, etc.

We consider $\mathscr{L}$, a set of **literals**, which is the disjoint union of all the above-mentioned sets. $\mathscr{L}$ also includes the empty label, denoted ϵ.

Moreover, we settle a **directed graph** as being a set of *nodes* N, connected by a set of *edges* $E = (N \times N)$. A node may carry a *label*, belonging to $\mathscr{L}$, and it is uniquely defined in the graph by an *identifier*, *id* in short, defined in $\mathbb{N}$. Similarly, an edge may carry a *label* (also belonging to $\mathscr{L}$), and is also uniquely defined in the graph by an identifier (defined in $\mathbb{N}$). A node is said to be:

- A **root node** when it has no incoming edge;
- A **structural node** when its label is in $\mathbb{I}$;
- A **leaf node** when it has no outgoing edge;
- An $\mathscr{L}$**-leaf node**, or simply a value node, when that node is a leaf and its label is in $\mathscr{L} \backslash \mathbb{I}$.

Two main graph data models have been studied so far: RDF graphs and property graphs. We will discuss them and the corresponding query languages (Section 2.6, respectively, Section 2.7). Only very recently, proposals to unify (or simplify the translation between) these two models have started to be investigated [112], focusing mainly on faithfully translating each aspect supported by each standard into the other. These efforts are still ongoing.

The main notations introduced above are summarized in Table 2.1.

2.2 Entity-Relationship model

The Entity-Relationship model [145] is a widely used way to model data when designing relational databases (see Section 2.3). It relies on the following concepts. An **entity** is an object of the real world, e.g., Inria Saclay lab, and an **entity set** is a set of *similar* entities, e.g., all French computer science labs. Each entity set has **attributes**, which are properties of the objects represented in the entity set. For instance, the Lab entity set may have "name", "address" and "SIREN" (company unique identifier) attributes. In the classical E-R model, attributes take their values in $\mathscr{L}$. Also, each entity set has a **key**, i.e., a minimal set of of attributes to uniquely identify an entity in the

Notation	Definition
Graphs	
G	a directed graph
N	the set of G nodes
E	the set of G edges
Nodes and edges	
ϵ	empty node, respectively edge, label
N_i	the G node of identifier i
N_i^l	the G node of identifier i and label l
$N_x \rightarrow N_y$	the G unlabeled edge connecting N_x to N_y
$N_x \xrightarrow{l} N_y$	the G l-labeled edge connecting N_x to N_y

Table 2.1: Summary of notations.

entity set. The "SIREN" attribute is a good candidate for being a key in the Lab entity. Entity sets may be connected with **relationship sets**. Two, three or more entity sets may be involved in a relationship set, they are referred to as binary, ternary and n-ary relationship sets (binary ones are most common). As for entity sets, relationship sets may have attributes. For instance, the entity set Student may be connected to the Lab one using a relationship set named "startsInternship", with a "date" relationship attribute to store when the student started her internship in the lab. A **relationship** is an instance of a relationship set, i.e., an actual connection between entities.

Further, the E-R model also features key and participation constraints. **Key constraints** state that every entity in a given entity set participates in at most one relationship, e.g., each Student is an intern in at most one Lab (no student is involved in two or more internships). **Participation constraints** allow to specify that every entity in a given set participate in at least one relationship, e.g., every Student is an intern in a Lab (no Student is left without an internship).

The ensuing E-R schema is a drawing representing entity sets as rectangles, attributes as ovals and relationships as diamonds. This is easily translated to a relational model (see Section 2.3).

Building on the above-mentioned E-R modeling, extended versions include features such as: weak entities, reflexive associations, specializations, generalizations and aggregations.

2.3 Relational data

As per the classical relational database theory [2], a *table* (or *relation*) consists of a *relation name*, of a *schema* and an *instance*. The *schema* is a set of *attributes*, each of which has a *name* (these names, denoted for instance $\{a_1, a_2, \ldots, a_n\}$, are all pairwise distinct), and an associated domain (as per Section 2.1). A relation's *instance* is a subset of the cartesian product of the domains that are part of the schema. Thus, each instance is a set of *tuples*. Within a relation, a subset of attributes $\{a_{i_1}, \ldots, a_{i_k}\} \subseteq \{a_1, \ldots, a_n\}$ may be *primary keys*, that is: (i) the values of these attributes suffice to uniquely identify a tuple of the relation, in other words, no two tuples agree on these values; and (ii) no subset of these attributes satisfies this property. Further, in a relation R, some attributes $\{a_{j_1}, \ldots, a_{j_l}\}$ may be *foreign keys* referring to a primary key of a relation S: this means that for every R tuple, there exists an S tuple whose primary key attributes are respectively equal to those

paper		
id	title	abstract
P1	RDF	W3C...
P2	XML	Data...
P3	JSON	Nodes...

wrote		
authorId	paperId	year
A1	P1	2023
A2	P1	2003
A3	P2	2013

author		
id	name	affiliation
A1	Alice	INRIA
A2	Bob	IPP
A3	Carl	IPP

Figure 2.1: Relational data modeling scientific publications and their authors.

of the foreign key in the R tuple.

For instance, relational data in Figure 2.1 encodes data in three tables (`paper`, `wrote` and `author`). In this example, `paper.id` is a primary key, and so is `author.id`. Two foreign keys refer to them: the `paperId` and `authorId` in the `wrote` table. Other constraints may be applied on tuples values, such as non-null values, year above 1900, etc. Such constraints need to be preserved when adding, modifying, or deleting tuples.

In practice, relational data comes in two flavors, of interest to us:

- Relational databases [145] follow the relational database conceptual model recalled above, and more specifically implement the ISO SQL language [169], standing for *Structured Query Language*. Among the best-known relational database management systems, there are PostgreSQL, Oracle, MySQL and Microsoft SQL server.
- The CSV data format (Comma-Separated Values) allows to model each table in a separate CSV file. Similarly, each relationship is modeled in a separate file. Moreover, attribute names can be lacking, i.e., the file may lack a header and just include data. Primary and foreign keys could be present in a file, or across several files, but there is no standard way to specify and describe them. As a consequence, data profiling is needed in order to identify candidate primary keys [1, 98].

2.4 XML documents

XML (Extensible Markup Language) [175] is a World Wide Web Consortium (W3C, in short) standard for describing structured documents. According to the standard XML Data Model [174], each document can be seen as a *tree*, whose nodes are of one of the following three types:

1. Document node (there is exactly one for each XML document);
2. Element nodes: one of them is the (only) child of the document node, while each other element node is the child of another element node; each element node carries a *name* (defined in $\mathbb{I}$), which is non-empty; a total order holds over the elements that are children of the same node. Each element may have zero or more attributes; each attribute has a name, which is non-empty, and one or several values (if there are several values, they are concatenated, separated with white spaces). There is no order among the attributes of a given XML node;
3. Text nodes, taking their values in $\mathscr{L}$, which are children of element nodes.

An XML document is presented in Figure 2.2 and its corresponding tree is shown in Figure 2.3. There are 10 elements, e.g., `document`, `author` and `title`; 7 text nodes, e.g., "`Alice`" and "`p1`"; and 4

```
<document>
  <authors>
    <author aid="a1" wrote="p1">
      <name>Alice</name>
      <affiliations>
        <affiliation>INRIA</affiliation>
      </affiliations
    </author>
  </authors>
  <papers>
    <paper pid="p1"/>
    <paper pid="p2">
      <title>RDF</title>
      <abstract>W3C...</abstract>
    </paper>
  </papers>
</document>
```

Figure 2.2: An XML document describing few authors and their publications.

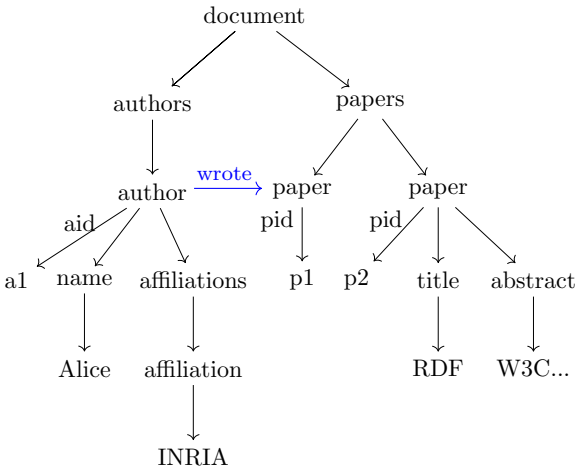


Figure 2.3: The XML tree corresponding to the data in Figure 2.2.

attributes, e.g., `id` and `wrote`. Following a frequent convention in XML data management research, the attributes are depicted as value nodes, and attribute names are shown as labels on the edge leading from an element to each of its attribute. Note that while edges connecting an element to its element children are unlabeled, each edge leading to an attribute is labeled with the attribute name.

On the contrary to relational databases where types are defined beforehand, e.g., in a relational schema, XML is self-describing data. This means that elements labels form the structure of the document; types and constraints *may* be added on top of it.

The first typing language introduced for XML is the **DTD** [168], standing for *Document Type Description*. This language allows to describe the children an element has. For instance, the XML document in Figure 2.2 may be accompanied with the DTD in Listing 2.1 (note that several, different DTDs may be valid for a given XML document). A statement of the form `<!ELEMENT element (child quantifier, [...])>` specifies that:

- `element` is an XML element label;
- `child` is the expected element child;
- `quantifier` is the frequency of that child expressed with a regular expression.

When an element accepts several children, pairs of `child` and `quantifier` are unioned with a comma. For instance, `<!ELEMENT document (authors, papers)>` states that the element labeled `document` contains exactly two child elements, `authors` and `papers`, each exactly once (the quantifier is omitted when the child appears exactly once); `<!ELEMENT author (name, affiliations?)>` states that an author element has (exactly) a name, and zero or one element `affiliations`. When the child is `#PCDATA`, this means that the element child is a text node and does not contain any further struc-

tured child. This is the case in `<!ELEMENT name (#PCDATA)>`. One can also specify element attributes, using a statement of the form `<!ATTLIST element attribute type value>`, where:

- `element` is the element name;
- `attribute` is the expected attribute for that element;
- `type` may be `#PCDATA` when the attribute value is a text node, an `ID` or `IDREF` (see below), etc.;
- `value` specifies whether that attribute is required (`#REQUIRED`), optional (`#IMPLIED`) or a fixed value (`#FIXED`).

It is worth stressing that, in a DTD, element label and element type are unified notions: each type corresponds to an element label; conversely, each element type corresponds to a label.

```
<!ELEMENT document (authors, papers)>
<!ELEMENT authors (author)>
<!ELEMENT papers (paper)>
<!ELEMENT author (name, affiliations?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT affiliations (affiliation*)>
<!ELEMENT affiliation (#PCDATA)>
<!ELEMENT paper (title?, abstract?)>
<!ATTLIST paper pid ID #REQUIRED>
<!ATTLIST author aid ID #REQUIRED>
<!ATTLIST author wrote IDREF #IMPLIED>
```

Listing 2.1: A possible DTD for the XML document in Figure 2.2.

To enrich XML schema descriptions, the **XSD** [177], standing for *XML Schema Description*, has been introduced and endorsed by the W3C. It allows to attach to elements simple and complex types, as well as type restrictions, thus bringing a richer typing system than in DTDs. For instance, Listing 2.2 gives a possible XSD for the authors described in the XML document in Figure 2.2. As opposed to DTD, XSD decouples the notions of element label and element type, i.e., each type corresponds to an element label, but the inverse is not always true. Therefore, element names (specified using a `name` attribute in a `<element>`) differ from element types, which may be simple or complex. Simple types include string, decimal, integer, boolean, date and time values (as in `type="integer"`). Complex types are described using the XML element `<complexType>`, e.g., an `affiliations` type is a list of zero, one or several `affiliation` elements, each being a string.

```
<element name="author">
  <key name="aid" type="integer"></key>
  <keyref name="wrote" refer="pid"></keyref>
  <element name="name" type="string"></element>
  <element name="affiliations">
    <complexType>
      <sequence>
        <element name="affiliation" type="string" minOccurs="0" maxOccurs="unbound"/>
      </sequence>
    </complexType>
  </element>
</element>
```

Listing 2.2: A possible XSD to describe authors depicted in Figure 2.2.

```
{
  "name": "Alice",
  "aff.": "INRIA",
  "wrote": [
    "JSON",
    {
      "title": "RDF",
      "abstract": "W3C..."
    }
  ]
}
```

Figure 2.4: A JSON document describing an author and her publications.

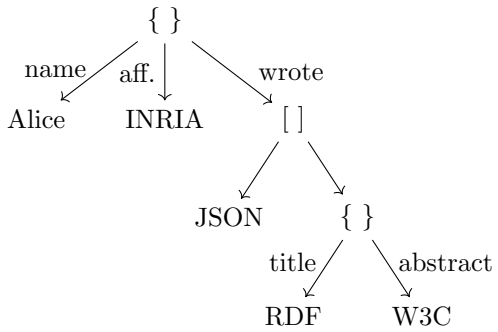


Figure 2.5: The JSON tree corresponding to the data in Figure 2.4.

Furthermore, DTDs and XSDs allow to declare identifiers, that are later used in references, in order to create connections between elements residing in different places of the document. Such connections are known as **ID-IDREF connections**. An ID allows to uniquely identify an element in the document (in the sense of a primary key in the database community). An IDREF is a reference to the element to refer to; they can be viewed as foreign keys. For instance, in Figure 2.3, the author of id "a1" wrote the paper of ID "p1". This information is encoded in both the DTD and the XSD. In the DTD, the paper and author ids are declared as `ID`, while `wrote` is an attribute that refers to some ID in the document. Note that nothing in the DTD constraints the IDREF values to be part of a given element ID, e.g., that `wrote` IDREFs refer to paper IDs. For the XSD, the author is declared as an integer in a `key`; similarly, the paper is declared as an integer key. The `wrote` attribute is said to be a `keyref` of a paper id. When not provided along the data, such connections may be inferred from the data itself with the help of data profiling [1, 98].

Finally, such schema descriptions are very convenient to validate a data file against a schema. Any data element that does not follow the type, attributes, children, or restrictions specified in the schema make the validation fail.

XQuery (XML Query) [176] is the standard language for querying and updating XML. Prominent current XML data management systems supporting XQuery include BaseX and eXist.

2.5 JSON documents

JSON [139] (JavaScript Object Notation) is a more recent, highly popular model to describe heterogeneous semi-structured data. JSON documents can be seen as trees, where a node may be:

- A *map*: a map node has the empty label ϵ, and has one or more elements, each of which is has a key (name from $\mathscr{L}$) and a value (which is a node).
- An *array*: each array node is also labeled ϵ, and has zero or more children, which are nodes. If there are children, a total order holds among them (in other words, they are indexed on $\mathbb{N}$).
- A *value* (string): this belongs to $\mathscr{L}$.

Figure 2.4 depicts a JSON document describing Alice, and her two publications, namely "JSON" and "RDF". Figure 2.5 shows the corresponding JSON tree. There are 2 maps (the outer element and the map for "RDF"). The former contains 3 keys (`"name"`, `"aff."` and `"wrote"`) and 3 values (`"Alice"`, `"INRIA"` and the array). The array contains one value (`"JSON"`) and one element, namely a map (the latter map), describing Alice's second publication.

As of today, there is no standardization for a JSON query language. However, the W3C drafted a proposal to create an XQuery-like language for JSON: this is JSONiq [171]. Moreover, JSON is extensively used in NoSQL databases, such as MongoDB and CouchBase. Note that these NoSQL databases adopt a different paradigm to store and query the underlying data. MongoDB sees data as a *collection* of *documents*, leveraging the JSON embedded structure, which can be queried using CRUD MongoDB-specific operations, e.g., the operation *find* corresponds to a full selection in SQL. Inversely, CouchBase fuses the strengths of relational databases such as SQL and ACID transactions with JSON flexibility and scale that defines NoSQL.

2.6 RDF graphs

RDF (Resource Description Framework) is the W3C recommended model [172] for sharing data on the Web; an RDF dataset is naturally viewed as a graph. The RDF standard relies on:

- *Resources identifiers* take their values in $\mathbb{U}$. A resource should be used to model an entity, thing, or person of the real world, for which a full identifier is known.
- *Blank nodes* are anonymous resources, for which a URI is not available. Instead, a blank node is described by an ID that only allows to identify it within the enclosing RDF dataset.
- *Literals* are string values and take their values in $\mathscr{L}$;

A **triple** is of the form $\langle$`subject`$\rangle\langle$`predicate`$\rangle\langle$`object`$\rangle$ where $\langle$`subject`$\rangle$ and $\langle$`predicate`$\rangle$ are URIs, while $\langle$`object`$\rangle$ can be a URI or a literal. For instance, Figure 2.6 depicts an RDF graph containing nine triples. A resource can be assigned zero, one or more types, using the special property `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` (for conciseness we refer to this URI as `rdf:type`). For instance, in Figure 2.6, the resource corresponding to Alice Dupont is declared to be of type author (see the fifth triple). A set of triples can be seen as an **RDF graph**, where each triple is a directed edge leading from the node corresponding to its subject, to a node corresponding to its object, as depicted in Figure 2.7.

One can also add *semantic information* to an RDF graph, describing relationships that hold between its classes and properties. Such relationships are stated as RDF triples using special properties, and including these triples in the RDF graph itself. For instance, in Figure 2.6, the triple $\langle$`yago:Paper`$\rangle\langle$`rdfs:subClassOf`$\rangle\langle$`yago:Artwork`$\rangle$ states that the class `Paper` is a specialization of the (generic) class `Artwork`. The set of semantic triples in a graph is commonly known as its *ontology*. Ontology triples are shown in dotted edges in Figure 2.7.

Several *ontology languages*, i.e., sets of dedicated properties for expressing specific relationships between classes and/or properties, have been standardized so far, having different expressive powers. **RDFS** [162], standing for *RDF Schema*, relies on four main properties, including subClassOf illustrated above. Further, RDF Schema allows stating that any subject of a given property is automatically considered of a certain type, using the *domain* (`rdfs:domain`) property. Symmetrically, the *range* (`rdfs:range`) allows stating that any object of a given property is of a certain type. For

```
1 <yago:AliceD> <yago:name> "Alice Dupont" .
2 <yago:AliceD> <rdf:type> <yago:Author> .
3 <yago:AliceD> <yago:wrote> <yago:Paper1> .
4 <yago:AliceD> <yago:wrote> <yago:Paper2> .
5 <yago:Paper1> <rdf:type> <yago:Paper> .
6 <yago:Paper2> <yago:title> "JSON fundamentals" .
7 <yago:wrote> <rdfs:domain> <yago:Author> .
8 <yago:wrote> <rdfs:range> <yago:Paper> .
9 <yago:Paper> <rdf:subClassOf> <yago:Artwork> .
```

Figure 2.6: RDF triples depicting an author and her two papers, and few RDFS ontology triples.

instance, in Figure 2.6, Lines 7 and 8 model the fact that a resource of type Author (may) write resources typed as Paper. One can also define a hierarchy among the properties, using the predicate `rdfs:subPropertyOf`. Two more predicates present in the RDFS vocabulary are: `rdfs:label` and `rdfs:comment`, to provide a human-readable name, respectively description, of a resource.

Among the more expressive ontology languages, OWL [191] allows to express more complex relationships between types and properties. For instance, unlike RDFS, it allows to state that two types are disjoint, i.e., that no resource can be simultaneously of two types. This leads to a question of *validity* of a graph with respect to an OWL ontology, a question that does not concern graphs with RDFS ontologies (since the latter does not allow expressing any constraints that the graph could violate). OWL also allows to define *derived types* by intersections/unions/set differences over other types, etc. *In this thesis, we will not consider OWL further, and rely on RDFS ontologies only.*

The triples of an RDF graph may be split in three categories:

- *Type triples* are triples whose property is `rdf:type`;
- *Schema (or ontology) triples* are triples using a property from the RDFS vocabulary;
- All other triples are *data triples*.

The presence of an ontology leads to *implicit (or inferred)* triples. First, two ontology triples may lead to a third one, e.g., if ⟨`c1`⟩⟨`rdfs:subClassOf`⟩⟨`c2`⟩ and ⟨`c2`⟩⟨`rdfs:subClassOf`⟩⟨`c3`⟩, it follows that ⟨`c1`⟩⟨`rdfs:subClassOf`⟩⟨`c3`⟩. Second, a data triple and an ontology triple may lead to a new data triple. For instance, if ⟨`x2`⟩⟨`wrote`⟩⟨`x1`⟩ and ⟨`wrote`⟩⟨`rdfs:range`⟩⟨`yago:Paper`⟩, then it follows that ⟨`x1`⟩⟨`rdf:type`⟩⟨`yago:Paper`⟩ (see the blue edges in Figure 2.7). The process of inferring triples from a graph based on its ontology is also called *reasoning*. The process of inferring all possible triples from a graph, adding them to the graph, and inferring again until no new triples can be found is called *saturation*. For RDFS, it has been shown [78] that the graph saturation can be computed in finite time (polynomial time complexity). Other more expressive ontology languages may lead to an infinite saturation process, e.g., if we allow an ontology rule such as "any Person has a friend that is also of type Person". In this thesis, we consider that the RDF graphs we work with have been fully saturated.

SPARQL [173], standing for *SPARQL Protocol and RDF Query Language*, is the W3C recommended language to query RDF graphs. Major RDF databases, also known as *triple stores*, include AllegroGraph, Virtuoso and StarDog.

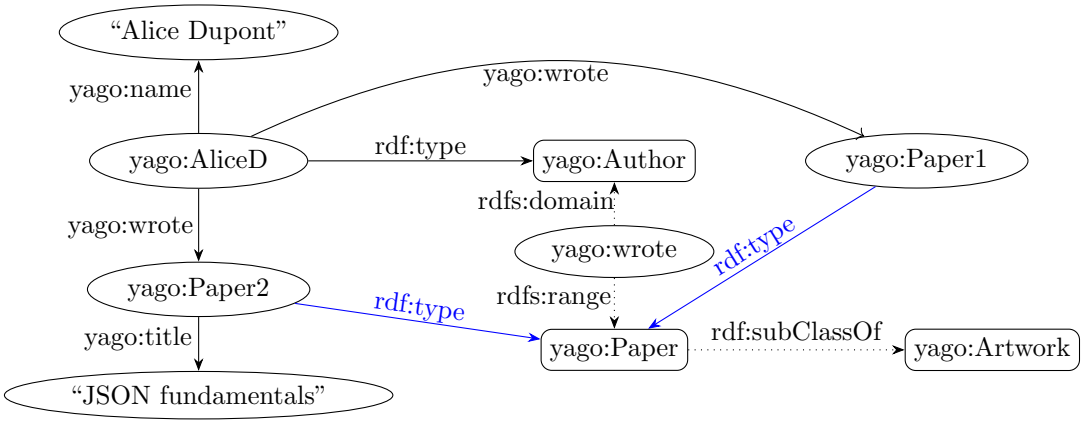


Figure 2.7: The RDF graph corresponding to the RDF triples in Figure 2.6. The graph also contains those inferred by the graph saturation (blue edges).

RDF has been used to describe numerous useful datasets. Many are *domain-specific* and focus on a given topic, e.g., ESKG, the NASA knowledge graph for Earth Science. Others, such as Yago [138], DBPedia [18] and WikiData [161], do not focus on one topic, but instead serve as repositories of universal knowledge. RDF graphs are also called Knowledge Bases (KBs) or Knowledge Graphs (KGs), highlighting their ability to represent data and knowledge.

2.7 Property graphs

In **property graphs**, information is organized in nodes, relationships, and properties:

- A node describes a structured record. It may have zero, one or few labels playing the role of *types*. A node may have a set of properties, where each property is a key-value pair. Nodes with the same set of labels may not have the exact same set of properties.
- A relationship is a directed labeled edge between a source node and a target node. Also, they may have properties, like nodes.

For instance, Figure 2.8 features a property graph describing three scientific publications and three authors. The nodes referring to publications are labeled as PAPER, while author nodes are labeled as AUTHOR and/or PERSON. Each node has a set of properties, in the attached rounded rectangle. Relationships are connecting authors to their publications using WROTE edges. Some of them have a property indicating the publication year.

No standardized schema for Property Graphs exists at the time of the writing. Each competitor in the field has its own way to define it. Recently, [14] proposed PG-schema, a formal definition of what a Property Graph schema could look like. They emphasize three main components: types, attributes and relationships between types.

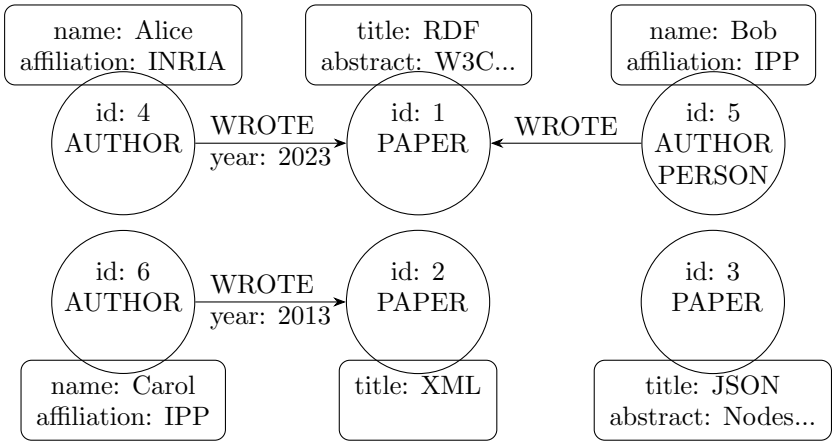


Figure 2.8: Property graph modeling scientific publications and their authors.

2.8 Summary

This chapter provided a brief recall of the main data models considered by the research work of this thesis: structured (relational) data, as well as four leading semi-structured data models: XML documents, JSON documents, RDF graphs and Property graphs. As shown above, these models feature a great deal of surface differences: tuples, trees or graphs, ordered or unordered, different notions of type, etc. Our goal in the rest of the thesis will be to bridge this variety through a common understanding of the data.

3

Related work

Chapter Outline

Chapter Abstract. In this chapter, we review existing works in the area of heterogeneous data integration (Section 3.1), data summarization (Section 3.2) and data querying (Section 3.3). We also discuss the limitations of existing works, as well as the existing gaps in those research areas. Finally, we position this thesis with respect to these existing works (Section 3.4).

3.1 Heterogeneous data integration

When working on real-world data-driven applications, users often need to deal with several datasets. Such datasets may be gathered from different providers, which did not coordinate when designing or producing it. Therefore, **data heterogeneity** may appear at several levels:

- The **data model** may differ because each data producer or production scenario may have different preferences. Thus, datasets may be available in the relational form, or as XML or JSON documents, as RDF graphs, as text, etc.;
- The **schemas** of different datasets (when schemas are available), or more generally, the **data structure** may differ, e.g., a dataset about students may describe them by their full name and student number, while another database provides the students' full name, home university and international student number.

Data integration systems aim at providing a unified interface to access, process and query a set of diverse, and potentially heterogeneous, datasets. When the data volume is small, one can attempt to work with heterogeneous data sources "as they are", manually gathering information about a few items of interest present in the data. This is, for instance, the case of data journalists: they work with multiple data sources (such as political speeches, city- and country-level data, etc.) but have no particular IT skills, and work under resource and time constraints. Following the recent emergence of data journalism [36], where journalists are increasingly aware of the opportunities provided by digital data and its associated processing tools, journalists are becoming interested in automated data integration systems, which could allow them to inspect and query the data with the help of friendly interfaces.

In larger organizations, e.g., companies or institutions, data integration systems (such as Tsimmis [73], Agora [123], Pegasus [7], Obi-Wan [40] and ConnectionLens [13]) have been designed to provide means to automatically integrate and manage such heterogeneous data. The interest and challenges raised by data integration remain quite high; they have been studied in different types of architectures. The three main categories are: mediator systems (described in Section 3.1.1), data warehouses (presented in Section 3.1.2) and data lakes (introduced in Section 3.1.3). Figure 3.1 highlights their main components, thus emphasizing their similarities and differences. We will describe these in Section 3.1.4.

3.1.1 Virtual integration with mediators

Data mediation has been initially introduced in [166]. The goal was to provide a unified architecture interacting with several data sources, with possibly different data models and schemas. As stated in Chapter 1, heterogeneity is still a main aspect of Big Data today (recall "variety"); sources may also evolve independently (recall "variability"). To enable data source use despite these challenges, mediator systems [73, 123, 40] provide (*i*) an integrated way for users and applications to interact with the underlying sources; and (*ii*) access to the data from its original source, also ensuring freshness. The mediation approach has also been termed *virtual data integration systems* in the literature.

Given a set of data sources, each having its own (local) data model and/or schema, a mediator system is composed of:

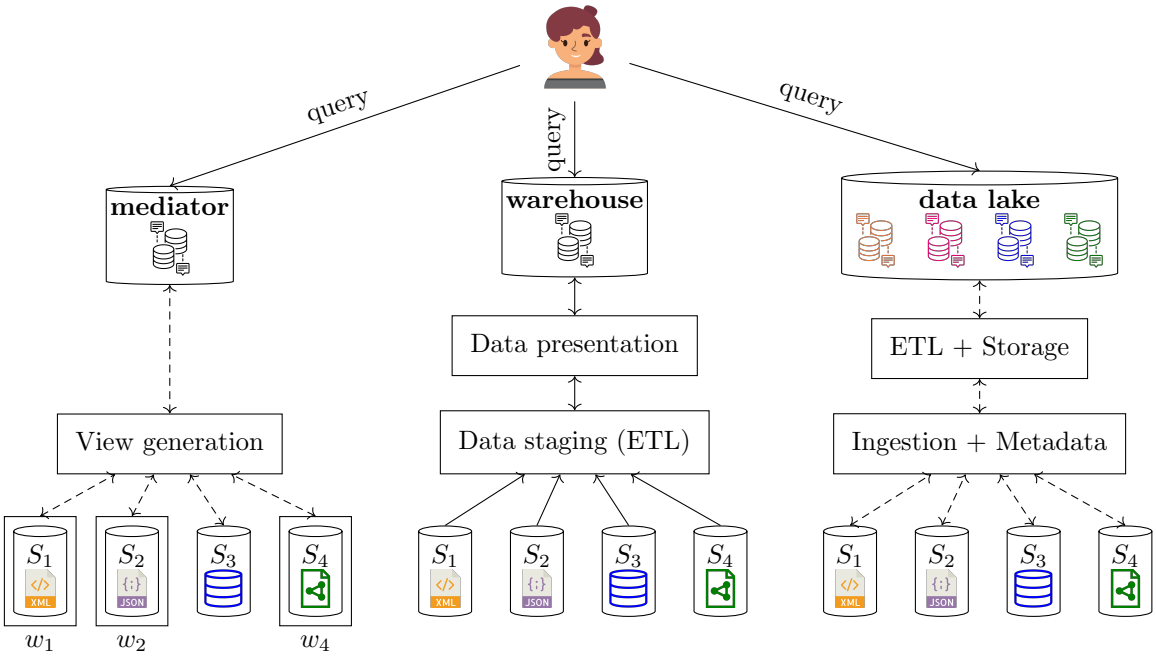


Figure 3.1: The three main architectures for heterogeneous data integration.

- A *global* (or *mediated*) *schema*, i.e., a unique and integrated schema exposed to users and applications;
- A set of *semantic mappings* specifying how local schemas and the global one relate;
- A *global* (or *meditated*) *data model* providing a unique formalism to interact with underlying data sources;
- A set of *wrappers* converting (local) data formats of the original sources into the mediator data model.

Finally, users and applications may send queries to the mediator system, expressed against the global (or mediator) schema. The mediator is in charge of processing the queries, possibly enlisting help from the data sources.

Global schema

The global schema is crucial to the functioning of a mediation system. To design it, IT experts and domain experts typically collaborate, aiming to ensure that it is sufficiently rich to be useful to users, and well designed to allow query optimization. The task of designing such a model is often tedious because each local source has its own (local) schema, thus inconsistencies between the different schemas have to be solved, domain and ontology differences have to be clarified and entity resolution may be useful to understand how similar entities are represented in each local schema.

In the upcoming examples, we will consider the following global and two local schemas. The global

schema has two relations: the first one records the projects researchers are working on; the second one stores the scientific field of each project. The local schema of the first source consists of a unique relation depicting the interest of some people in a field. The second source has two relations: the first one stores data about researchers' grants; the second one relates grants to projects. We will denote the global schema relations as `G.Work(R,P)` and `G.Area(P,F)`. Local schema relations will be denoted by `S1.Interest(Pe,F)`, `S2.HasGrant(R,G)` and `S2.For(G,P)`.

```
Global schema
  Work(Researcher, Project)
  Area(Project, Field)

Local schema for source 1
  Interest(Person, Field)

Local schema for source 2
  HasGrant(Researcher, Grant)
  For(Grant, Project)
```

Semantic mappings

For a given set of n local schemas $S_1, \ldots, S_n$, each of one relation, and a global schema of m relations $G_1, \ldots, G_m$, **semantic mappings** specify how local and global relations connect to each other. A mapping is a logical formula of the form A_i `:-` B_i where A_i and B_i are queries over the global or the local schema(s).

Depending on the mapping structure, we identify three kinds of mappings:

- **Global-As-View**, or GAV, mappings express the global schema relations *in terms of* local schemas relations, as in `G.Work(R,P) :- S2.HasGrant(R,G), S2.For(G,P)` and `G.Area(P,F) :- S1.Interest(Pe,F), S2.HasGrant(Pe,G), S2.For(G,P)`. Each global schema relation G_i can be seen as a view of some local relations, thus a GAV mapping states how it can be computed from the sources. That is why GAV mappings are straightforward to implement, thus leading to its wide adoption [73, 26]. However, the update or integration of sources is complex. Indeed, when a local source S_i is updated, all mappings using S_i need to be updated with the new source definition. Similarly, when a new source is added, new attributes may need to appear in the global schema. Therefore, mappings whose definition has changed to include the new attributes need to be updated.
- **Local-As-View**, or LAV, mappings express the local schemas relations *in terms of* global schema relations, as in `S1.Interest(Pe,F) :- G.Work(Pe,P), Area(P,F)`. Note that local relations `HasGrant(R,G)` and `For(G,P)` cannot be expressed in terms of the global schema as it does not contain grant information. LAV mappings describe each data source as a view over the mediator schema. However, LAV mappings do not suffer from the GAV main problem related to changes in the data sources: they can be integrated independently from others since each LAV mapping does not refer to other sources schemas but only to the global schema, as emphasized in [114, 146]. The price for such flexibility is to have more complex rewritings, as we discuss below.
- **Global-Local-As-View**, or GLAV, generalizes both GAV and LAV as follows. It associates a query over the global schema to a query over the local schemas, as in `Interest(R,F) :-`

`Work(R,P), Area(P,F)` and `HasGrant(R,G), For(G,P) :- Work(R,P)`. GLAV mappings generalize both GAV and LAV mappings. They maximize the flexibility and the expressive power of data integration. GLAV mappings are used in systems such as [40, 42].

Wrappers

When a mediator integrates data sources that (only) differ in their schemas, a global schema and a set of semantic mappings suffice to interact with the underlying data. However, when sources having different data models, such as relational, XML, JSON, or RDF, are integrated together, **wrappers** are needed. They allow to "wrap", or "hide", the underlying data model and provide to the mediator the data *as if* it was in the mediator data model. Many mediators [73, 123] consider a relational mediator schema; more recent ones use RDF [40] at the mediator level, for more flexibility, and the possibility to use ontologies to describe, and enrich, the mediator schema.

Query answering in a mediator system

Typically, when a user asks for a *global query*, i.e., a query over the global schema, the mediator has to (*i*) translate that query into local queries referring to the original data sources schemas, such a task is called *query rewriting* [85]; and (*ii*) build a global query to aggregate local query results.

In a *GAV setting*, the rewriting task is easy because the global schema is expressed in terms of local ones. Therefore, a user query is simply rewritten by **query unfolding**. This replaces each term of the global query by its local schema twin using semantic mappings. For instance, suppose the user asks for all fields by issuing the following query: `q(F) :- G.Area(_,F)`. With query unfolding, this query is rewritten as `q(F) :- S1.Interest(Pe,F), S2.HasGrant(Pe,G), S2.For(G,_)`, meaning that asking for `q(F)` is equivalent to asking for something (_) that received a grant `G` such that this grant has be given to a researcher `Pe`, working in some field `F`. GAV query rewriting has low complexity (linear in the size of the mappings and given query).

In a *LAV setting*, the task is more complex because the mappings express how the sources can be described by the global schema. Therefore, the global query cannot be directly translated. Instead, an inference mechanism is needed to re-express each term in the global query using local schemas terms. There exist three main query rewriting algorithms in this case: Bucket [114], Minicon [141] and Inverse-rules [60]. For instance, if a user asks the query `q(F) :- G.Area(_,F)`, the query rewriting component will have to find out that `S1.Interest`, `S2.HasGrant` and `S2.For` will be useful. Then, it will build multiple join combinations of these views, check which of these are guaranteed to provide query results, then build the rewriting as a union of all these. In a LAV setting, complete rewritings may not exist, i.e., the computed results may be just a subset of those that the user query is asking for. However, in such cases, we seek *maximally contained* rewritings, that is, the largest subset of the query result that can be computed based on the available view definitions. LAV query rewriting is NP-hard (in the size of the query plus the size of the views) for simple, conjunctive relational queries and views [114].

In a *GLAV setting*, query rewriting starts with a LAV step, rewriting the query formulated against the global schema, with the "views" defined by the global schema queries of the existing mappings. This process leads to reformulating the query over a set of atoms, each of which corresponds to one mapping. Then, a GAV rewriting step is applied, replacing each of these atoms with the corresponding source query from each mapping, leading to the desired outcome: a rewriting of the user query over the data sources. The complexity of this process is dominated by the first step

(LAV rewriting). Overall, GLAV (and GLAV rewriting) correspond to the most complex setting, and also the one providing the largest expressive power.

3.1.2 Physical integration with data warehouses

The warehouse architecture has been firstly mentioned in [54]. The goal was to design a "*business data warehouse*" to provide users a *decision support* system where they can focus on the information, rather than on how to obtain it. To achieve this goal, a data warehouse is a unique consolidated repository where data, coming from several heterogeneous sources, is replicated, cleaned, consolidated, and stored. A data warehouse enforces a **unique schema** that users are familiar with and based on which they can query the data of interest to them. The main advantage of a consolidated repository is to deliver high performance since processing is upstream and not done at the query time. It also adopts a **unique model**: relational or multi-dimensional [50, 108].

In a typical data warehouse, data is stored in relations, one per fact (recall Chapter 2). Relational modeling allows very fast retrieval, thanks to good indexation strategies and relational optimizers. However, they are less performant when it comes to specify many conditions. Therefore, **dimensional modeling** has been introduced in [187] and is now widely used. Data is seen as a set of *facts*, to which are associated a set of *dimensions*, as well as a value along each dimension, and a (set of) *measure*. Such data is typically called a *n-dimensional cube*. For instance, in a retail company, a cube describes some Products (x axis) sold by a Store (y axis) at a given Time (z axis) for a given price (the numerical measure in the cube cell). A data warehouse can be queried with the help of operations specific to this multidimensional model, e.g., *slice*, *dice*, *drill-down*, *roll-up*, etc., well described in specialized textbooks [108].

From the system organization viewpoint, with our focus on data integration, in a data warehouse, we identify:

- The **operational systems area** contains the (external) data sources, which are self-ruling, generate data mostly outside of the warehouse' realm of control, and are not coordinated with other sources;
- The **data staging area** provides an ETL (*Extract-Transform-Load*) pipeline, composed of the three following steps:
  1. **Extraction** gets data into the warehouse by (i) reading and understanding the sources; then (ii) copying necessary data in the staging area;
  2. **Transformation** applies various data cleaning processes, e.g., values normalization, deduplication, missing values inference, then transforms the cleaned data into dimensional tables;
  3. The **loading** step sends the cleaned dimensional tables to the storage, indexes them and notifies users that new data has arrived in the warehouse.
- The **data presentation area** is where data is made available for direct querying by users and analytical applications. This is the farthest that they can go in the warehouse; they are not allowed to enter the operational systems and data staging areas. The presentation area is often composed of data marts, i.e., small portions of the whole warehouse.

Data warehouses, such as those proposed by Amazon [179], eBay [180] or Google [186], are widely used by companies for analytical queries, data mining, business statistics, etc. Indeed, they are

well-suited for specific, targeted contexts where large amounts of data can be (*i*) integrated in a central repository, (*ii*) organized along dimensions, and (*iii*) queried intensively with analytical queries.

3.1.3 Schema-less data integration

In recent years, settings where very large number of data sources need to be used together has led to new paradigms for data integration. Unlike the mediator and the warehouse architectures, in very large-scale data integration it becomes unfeasible to establish a global schema, due to the high number of sources and their heterogeneity. Therefore, new platforms have emerged for this task; we discuss their main characteristics below.

Data lakes have emerged in the last decade, sample systems being [10, 111, 83, 25, 13, 203, 68]. Since the area has not completely converged yet, many data lake definitions co-exist, depending on the research area, application domain and user needs. We focus on the definition provided in a recent survey [84]: "a data lake is a flexible, scalable data storage and management system, which ingests and stores raw data from heterogeneous sources in their original model, and provides maintenance, query processing and data analytics in an on-the-fly manner, with the help of rich metadata.". A data lake also often comprises sources which have different schemas. The data lake itself is *schema-less*, in the sense that no unified schema is sought or built. Some data lakes propose to unify the different formats used by each source, e.g., in a relational [83, 25] or graph paradigm [13, 203]; however, most of the data lakes do not require it. We now describe the main aspects of a data lake, following the recent survey [84] and a recent VLDB tutorial [132]. We first discuss data ingestion, then metadata extraction. We continue with data processing and data storage. Finally, we cover the exploration of data lakes by users.

Data ingestion

During the **ingestion phase**, source data is loaded in the data lake. This (raw) ingested data is hard to use as is, given that the structure and semantic of the data is not known. At this point, the data lake is simply a "data swamp" of several datasets co-existing in the same place. Therefore, the next step is crucial: the acquisition of as much *metadata* as possible.

Metadata extraction

To be profitable, a data lake should follow the *FAIR principles*: data should be Findable, Accessible, Interoperable and Reusable. This is why metadata extraction is primordial and should be designed using a **rich and flexible metadata model**, which also allows to further access and query the data. Three main metadata models exist:

- *Generic models* [13, 83, 25] store metadata based on their type, e.g., structural, semantic, content-based;
- *Data vaults* [134, 75] define hubs (to model business concepts), links (to connect hubs) and satellites (to attach information to hubs and links);
- *Graph-based models* [203, 68] represent metadata using networks or hyper-graphs. This mainly allows to connect datasets based on their common metadata nodes.

More specifically, **metadata extraction** allows to discover the structural, semantic and provenance information of the loaded datasets as follows:

- *Structural metadata* often takes the form of schemas extracted out of each source (see Section 3.2 for more details on schema extraction). For instance, CONSTANCE [83] leverages existing schemas that may accompany the data, such as DTDs and XSDs for XML documents or relational schemas for databases. Lacking these, CONSTANCE extracts schemas from semi-structured data by looking for self-describing data attributes, e.g., headers for spreadsheets, and "has-a" relationships. DATAMARAN [72] builds "structure templates" in the form of regular expressions built on the data instances. As of now, no data lake proposal extracts complex structural metadata (as those described in Section 3.2) from semi-structured datasets. When datasets have schemas, in particular relational datasets, schema matching [144] and/or schema mapping [65] may be applied to find how they relate and possibly merge them on their common attributes. Also, *structural enrichment* is possible, for instance, to discover functional dependencies that a dataset either fully satisfies, or "mostly" satisfies, i.e., there are a few violations.
- *Semantic annotations* can be added to the data, e.g., frequent keywords and Named Entities, that may be connected to an external knowledge base and/or contextualized and disambiguated using entity linking [38];
- *Data provenance*, also mentioned as *data lineage*, refers to the data origin and possibly additional information about the data suppliers. For instance, GOODS [86] is a provenance-first lake where data can be explored based on its provenance, which is composed of the data provider and how it relates to other datasets (which datasets depend on it / it depends on).
- Some metadata models also keep track of the time to provide *versioning*, as in GOODS [86].

As of today, data lakes often extract metadata for each data source, one at a time, and then fuse all the extracted content in a metadata model. However, transient value across the lake, i.e., metadata extracted when considering several, if not all, data sources, is not well explored yet.

Data processing

After metadata extraction, the lake still needs some **processing and maintenance** to perform data organization and/or cleaning tasks.

Dataset organization discusses how to structure, organize, and navigate the datasets in the lake. Mainly three categories can be identified for this task:

- *Catalogues*, such as in GOODS [86] and CONNECTIONLENS [13], store in tables various metadata, such as the provenance, date of acquisition, information about the data semantics, etc.;
- *Classification models* categorize datasets using classification algorithms, e.g., k-nearest neighbors algorithm in [11], where similar datasets are close to each other in the feature space, thus classified in the same category;
- *DAG-based methods*, as in [121, 131], organize datasets in Directed Acyclic Graphs where datasets are connected based on their common features.

Data cleaning seeks to discover and fix data quality problems at a schema and instance level. For instance, missing or erroneous values, redundancies, value (re)formatting should be identified, and

repaired. In general, the cleaning task is performed during the data processing. However, some authors point out that early cleaning, i.e., during data ingestion, as proposed in CLAMS [67], avoids error propagation, and increases the data quality at a lower cost.

Storage

Most data lakes store datasets in their original data model. For this **storage** task, the main approaches are as follows:

- *File-based storage systems*, as in CLAMS [67], mostly rely on HDFS (*Hadoop Distributed File System*), a distributed file system widely used in massively parallel data processing;
- *Single data stores* leverage a single database to store the data, this is the case for [163] and CONNECTIONLENS [13];
- *Polystore systems*, such as CONSTANCE [83], GOODS [86] or ESTOCADA [9] provide a single entry point to access many heterogeneous datasets;
- *Cloud data lakes* [200] store data in a cloud, such as those provided by IT giants: Azure Data Lake from Microsoft, AWS data lake from Amazon and Delta Lake from DataBricks, among many others.

Depending on the system size and capabilities, data lakes have been referred to as "data puddle", "data pond", "data lake" or "data ocean" in [80]. Recently, *data lakehouses* [16, 17] have been proposed as a hybrid approach between data lakes and data warehouses (recall Section 3.1.2). The goal is to add, on top of a data lake architecture, processing capabilities typically provided by data warehouses, in particular analytics.

More recently, the new term of *data mesh* [51] has been introduced in 2019. A data mesh is an architectural framework, referring first (or mainly) to how data is produced, shared, and reused (thus, less centered on the physical storage layer). The stated goal of data mesh is to integrate diverse sources in a single point of access to the data, while domain teams (that is, those actually performing the tasks of the company or organizations owning the mesh) keep their freedom to create, share, derive, reuse, and govern the data. Data meshes are not yet well studied in the literature because its is an extremely new concept. However, major companies are advertising their versions thereof, as in AWS lake formation from Amazon and IBM Data Fabric from IBM.

Exploration

Several kinds of users interact with the data lake: data scientists and business analysts, who apply models on data; information curators, who define new data sources and organize the data lake; and operations teams which maintain the data lake. **Data lake exploration** is challenging because data should be findable and usable by users (recall FAIR principles); it is also heterogeneous, with no consolidated schema. This is where all the metadata computation and (pre)processing show their importance. In general, users interact with the data lake in two ways: finding datasets related to an input dataset; or sending queries to a unified lake interface.

- To find *related datasets*, the goal is to, given an input dataset, return the top-k datasets in the lake that are the most similar, or joinable on the input dataset. This task is called *discovery of semantically related datasets* and compares attributes and their values according to some semantic similarities. For instance, JOSIE [202] finds similar tables by computing the overlap

between the input tables and the lake tables; the higher the overlap, the better. Discovery of *joinable datasets* methods find how different tabular datasets might be related by their columns. Columns may be related based on their label, cardinality, distribution and/or data values. For instance, PEXESO [57] finds semantically joinable tables by transforming textual values into high-dimensional vectors and computing their vector similarities. Note that all these techniques are designed for *tabular* data.

- The data lake may also provide a *unified querying interface* to access the heterogeneous datasets (see Section 3.3 for more on querying). For instance, CONSTANCE [83] helps users to explore the lake by browsing existing data sources and their metadata. Then users may write a query (in SQL for structured data, or JSON for NoSQL data) on a data source they are interested in. They may also run a keyword search over a data source or its schema. Finally, users can ingest their query results in the data lake (recall the importance of provenance and metadata). Similarly, COREDB [25] provides a unified interface through a REST API to query or send CRUD operations to the data lake. Queries are internally transformed into elastic search queries for full-text search, SQL queries for relational queries or SPARQL queries for knowledge graphs. CONNECTIONLENS [13] provides an efficient algorithm to query a graph-based data lake using keywords, where an answer is a sub-tree of the data graph, connecting one node that matches each query keyword.

Given the large number of datasets, as well as their high degree of heterogeneity, data lake exploration remains an active area of research.

3.1.4 Data integration architectures comparison

Mediators, warehouses, and schema-less integration all target the same goal: automatically integrating heterogeneous data under a unified user interface. Beyond this common goal, their differences and specific features can be summarized as follows:

- Mediators and warehouses follow a hierarchical, or vertical, architecture, with a global schema hiding local sources' schemas, while schema-less integration can be seen as "horizontal" (see Figure 3.1);
- Warehouses physically integrate the data in a unified schema (plain edges in Figure 3.1) while mediators and data lakes expose virtual views over the data (dashed edges in Figure 3.1). Thus, warehouses require refresh strategies to ensure recent data is queried. On the contrary, virtual integration guarantees the data read by an application is always fresh; however, query processing is more complex since queries are stated over a different schema than the actual storage schemas;
- In mediator systems, there is typically no data curation or transformation; the data is exposed as is;
- With respect to their handling of data heterogeneity: warehouses eliminate it by migrating data in the target data model; mediator systems handle it through dedicated wrappers (w_1, w_2, w_4 in Figure 3.1); data lakes work with data as it is in the sources or may apply a unified model.

No single data integration architecture fits all needs. Data warehouses are the architecture of choice when the main processing of the data is analytical, and the data is not too frequently updated.

Mediation architectures are suited to contexts where a relatively low number of data sources, each with high autonomy requirements, cooperate for a specific data sharing application. Schema-less data integration, and in particular data lakes, are used when the number and the variety of sources is too large to support designing an integrated schema.

3.2 Data summarization

Faced with a very large dataset, it may be difficult for users to learn about its content, as it is sometimes hard to even figure out what questions to ask. Data summarization builds structured and concise summaries out of semi-structured datasets; since a summary is relatively small, users can visually inspect it to gather first insights about the data. Many approaches have been proposed to automatically summarize semi-structured datasets. Summarization methods for XML documents are surveyed in [63, 128], for RDF graphs in [43, 101], and more generally for data graphs in [32, 119]. Regardless of the data model, existing summarization techniques can be divided in four categories:

- **Structural approaches** summarize a dataset structure by creating groups of nodes considered *equivalent*. Different algorithms rely on different notions of node similarity; generally, these notions leverage node labels, types, and/or their neighborhood, i.e., their incoming and outgoing edges. Note that, depending on the input data and the similarity notion, the summary may still be too large to be inspected by users. Such structural approaches have been used to summarize XML data [15, 79], RDF graphs [77, 76], Property Graphs [33], and JSON documents [20].
- **Pattern mining methods** leverage mining techniques to discover *patterns*, out of which the summary is built. Works in this line include [156, 204, 205].
- **Statistical approaches** [59, 92, 142] summarize data from a quantitative point of view. They often count class instances, property and value types, properties frequency, etc.
- **Hybrid methods**, such as [103, 133, 147], combine structural, mining-based and/or statistical methods.

Some summaries have also been called *a-posteriori schemas*, in opposition to the typical *a priori schemas*, traditionally defined before a database is populated. From this perspective, summarization methods can be related to **schema inference** techniques, even if the overlap is quite partial. Specifically, schema inference applies in settings where a specific schema language is available. Then, schema inference starts from a dataset and attempts to find a schema specification expressed in the given schema language, such that the dataset *is valid with respect to* (or *conforms to*) the computed schema. Often, many schemas satisfy this requirement; schema inference methods may then return one schema or another, depending on other objectives, such as generality, conciseness, etc. Schemas for semi-structured data are usually some form of grammar, which may contain regular expressions, etc., thus they are not suited for users lacking technical skills. Also, depending on the dataset, the target schema language, and the specific schema inference method, inferred schemas may be large, especially if the data instances exhibit some variability. Schema inference techniques have been devised for all semi-structured data models, including XML [48, 109], JSON [20, 19, 158], RDF [81] and Property Graphs [113, 143, 14].

Below, we describe in detail five structural summarization techniques, which we either adopt, or compare with, in this thesis. We first categorize structural summarization techniques in two groups:

quotient and non-quotient summaries (Section 3.2.1). We continue by describing label summarization (Section 3.2.2) and path summarization (Section 3.2.3). Then, we discuss RDF quotient summaries (Section 3.2.4). Next, we move to schema inference techniques for XML (Section 3.2.5), JSON (Section 3.2.6), and Property Graphs (Section 3.2.7). Finally, we discuss the challenges of devising novel techniques to summarize data of different models (Section 3.2.8).

3.2.1 Quotient and non-quotient summaries

Structural summarization methods can be divided in two categories:

- **Quotient approaches**, such as [77, 104, 126], build summaries on the following notion: nodes that are considered *equivalent* are grouped together, and this equivalence will serve to "compress" (or represent) the original graph into a graph-shaped summary. Formally, suppose a graph G and an equivalence relation $\equiv$ over the nodes of G. The equivalence relation *partitions* G nodes into *equivalence classes*, where an equivalence class contains $\equiv$-equivalent nodes. Thus, each node is represented by exactly one equivalence class. The quotient graph, or *quotient summary* or simply *summary*, of G by $\equiv$ is the graph having:
  - A summary node for each equivalence class of the $\equiv$ relation;
  - For each G edge of the form $n_1 \xrightarrow{a} n_2$, an edge $m_1 \xrightarrow{a} m_2$, where m_1, m_2 are the equivalence class nodes representing n_1, respectively n_2.
- **Non-quotient techniques**, as in [79, 102], structurally summarize datasets by other means. Contrary to quotient summaries, a non-quotient technique (i) may not represent all the nodes in the original data, and/or (ii) may represent a node of the original graph by more than one summary node.

Among the numerous summarization techniques, we will discuss the ones that we used in this thesis or are very close. Some of them are quotient approaches, while others are not.

3.2.2 Structural summarization based on labels

Label summarization groups structural, i.e., non-value, nodes with the exact same label into a group (equivalence class) to which we assign the same label. Values are grouped based on their parent label, i.e., all value children of equivalent parents labeled `a` are grouped together in an equivalence class labeled `#val`. For instance, the XML document given in Figure 3.2 describes four people and leads to the summary in Figure 3.3. All the four `person` nodes have been grouped together, regardless of their position in the XML tree. This is because they carry the same label. All people names are therefore grouped in the node `#val` whose parent is `name`. The same applies for people affiliations and addresses. Note the cycle `person` → `friends` → `friend` → `person`. It appears because `person` nodes originate from different places in the XML document.

In general, label summarization can be applied on any graph. However, it is useful only if some nodes have the same label. Otherwise, the label summarization will output the same graph as given as input. For instance, label summarization should not be applied on an RDF graph, where all node labels are unique.

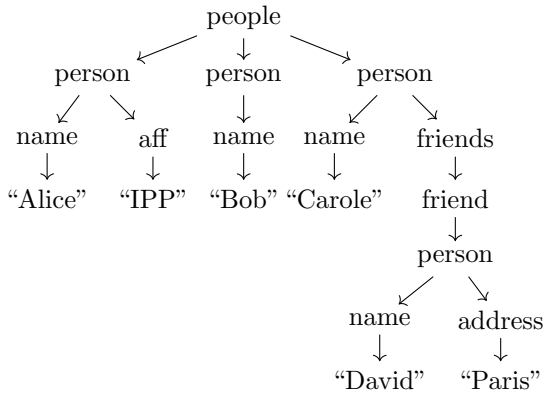


Figure 3.2: An XML tree describing four people.

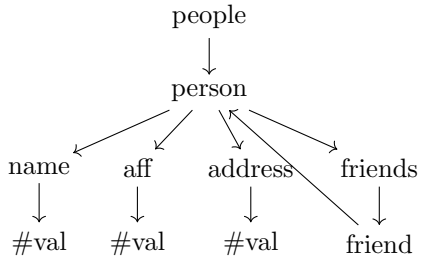


Figure 3.3: The label summarization of the XML tree in Figure 3.2.

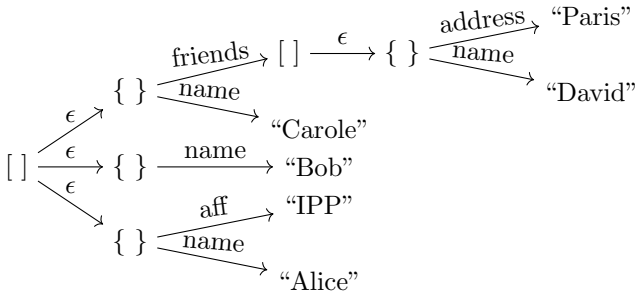


Figure 3.4: A JSON tree representing few people.

3.2.3 DataGuide summarization

The first summarization technique based on paths introduced in the literature is that of DataGuides [79]. A DataGuide summary can be built on a rooted acyclic graph; it groups nodes occurring on a common *path from the root*. Formally, given a rooted acyclic data graph G:

- A **path** is a sequence of nodes connected by consecutive edges, of the form $\{n_1, e_2, n_2, e_3, ..., e_i, n_i\}$, where n_i is the i^{th} node in the path and e_i the i^{th} edge in the path connecting n_{i-1} to n_i. For both n_i and e_i, $1 \leq i \leq depth(G)$. The **path label** is the set of dot-separated node and edge labels. For instance, the path { [ ], ϵ, {}, name } is denoted `[].`ϵ`.{}.name`.
- The **target node** of a path p is the last element of the path, i.e., n_i.
- A **path from the root** is a path whose first element n_1 is G root.
- A **DataGuide** D of G is a rooted graph such that every path from the root in G corresponds to exactly one path in D, and every path of D corresponds to a path in G. DataGuides contain no $\mathscr{L}$-leaf value since the goal is to reflect the input graph structure. Instead, different $\mathscr{L}$-leaf values occurring on the same path from the root are considered equivalent.

DataGuides were originally introduced to summarize Object Exchange Model (OEM) data, a semi-structured graph-structured data model issued from research carried at U. Stanford and IBM in the 90's [73]. DataGuide summarization easily translates to other semi-structured data models, thus in Figure 3.5 we show two DataGuides built from the JSON tree describing few people in Figure 3.4. In the DataGuide in Figure 3.5a, all the maps in the root array are grouped together as they appear on the same path rooted in the graph root, i.e., `[].{}`. The corresponding DataGuide map has for fields `friends`, `name` and `aff`, the union set of all the map fields in the three original maps (one with a name and a list of friends, one with a name and an affiliation, and one with a name only). For what concerns value summarization, the JSON paths `[].ϵ.{}.name.`"`Alice`" and `[].ϵ.{}.name.`"`Bob`" lead to only one path in the DataGuide: `[].ϵ.{}.name.#val`.

It is important to note that several valid DataGuides may exist for a given input graph. For instance, Figure 3.5a and 3.5b are both valid DataGuides for the data graph in Figure 3.4. In fact, the DataGuide in Figure 3.5b is *minimal*, while the other is not. Intuitively, **minimal DataGuides** compress the original graph as much as possible, by minimizing the number of nodes. Such compression can be achieved starting from a non-minimal DataGuide, treating it as an automaton, and applying *state minimization algorithms* [91].

Any graph summary, thus any DataGuide, induces some level of *information loss*. Among DataGuides, the minimum ones maximize such information loss. For instance, when looking at the minimal DataGuide only, one cannot distinguish the two lists (the graph root and the list of friends), respectively the four maps (the three maps of people and the one for Carole's friend).

As defined above, a DataGuide guarantees that a given path in D reaches a unique DataGuide node. However, nothing guarantees that a DataGuide node can be reached by only one path. For instance, in Figure 3.5b, one can access the element {} using, `[].{}`, or `[].{}.[].{}`, among others. To prevent this, **strong DataGuides** have been introduced. Let G be a rooted graph and p a path in G.

- The *target set* $T_G(p)$ is the set of G target nodes that can be reached by traversing p;
- $L_G(p)$ is the set of G paths having for target set $T_G(p)$;
- Conversely, $L_D(p)$ is the set of D paths having for target set $T_D(p)$;
- Intuitively, $L_G(p)$, respectively $L_D(p)$, contains more paths than only p when a node in $T_G(p)$, respectively $T_D(p)$, can be accessed by different paths.

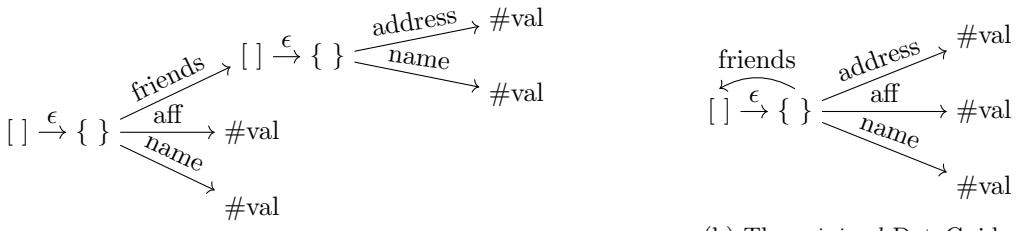
which can be formalized as follows:

$$T_G(p) = \{n_i \mid p \text{ is a path in } G\} \tag{3.1}$$

$$L_G(p) = \{z \mid T_G(z) = T_G(p)\} \tag{3.2}$$

$$L_D(p) = \{z \mid T_D(z) = T_D(p)\} \tag{3.3}$$

Based on these notations, the authors define **strong DataGuide** as a DataGuide where for every path $p \in G$, $L_G(p) = L_D(p)$.



(a) A possible DataGuide.

(b) The *minimal* DataGuide.

Figure 3.5: The two possible DataGuides for the JSON tree presented in Figure 3.4.

For instance, let p be { [ ], ϵ, {}, name }, G be the JSON tree in Figure 3.4, and D the DataGuide in Figure 3.5b. We first compute $T_G(p)$, which we refer to as T_G, as being { "Alice", "Bob", "Carole" } and $T_D(p)$, or simply T_D, as being { `#val` }. Next, we compute $L_G(p)$ and $L_D(p)$, respectively referred to as L_G and L_D. L_G corresponds to the set of every graph path z reaching the three nodes in T_G. So, $L_G(p) =$ { `[ ].ϵ.{}.name` }. L_D contains every DataGuide path z reaching the name values, i.e., { `[ ].ϵ.{}.name`, `[ ].ϵ.{}.friends.[].{}.name`, ... }. Because $L_G \neq L_D$, the DataGuide in Figure 3.5b is not strong. On the contrary, the other DataGuide (Figure 3.5a) is strong: there is only one way to reach each DataGuide node.

In general, *strong DataGuides* minimize the information loss but are heavy to store (because they are larger than DataGuides and minimal DataGuides). On the contrary, *minimal DataGuides* are the cheapest to store, but also introduce approximations. Also, they are harder to maintain: when a new node needs to be introduced, most of the DataGuide may have to be revisited. This because of the approximations, which make the task of finding where to add the node and how to connect it to others harder. In this thesis, **we prefer strong DataGuides.** Finally, note that a strong DataGuide is also minimal if each node label appears at most once. Indeed, node labels appearing in different places in the (strong) DataGuide allow minimization.

3.2.4 Structural RDF quotient summaries

Novel summarization techniques leveraging quotient graphs have been introduced in [77], where authors propose two kinds of summaries, each derived in two flavors. Those summaries both rely on the labels of the edges incoming and outgoing the RDF graph nodes; they are respectively called *weak* and *strong* summaries. The first summarization flavor consists of grouping resources first, based on their adjacent properties, then attach type information, if present. The latter makes RDF types first-class citizens and uses edge neighborhood only on nodes for which no type information is available.

Figure 3.6 depicts an RDF graph describing four people, namely Alice, Bob, Carole and David. The first three are typed as people while David is typed as a Friend. Each person has its own set of properties, e.g., Alice has a name and an affiliation while Bob only has a name. The RDF graph also includes a (dashed) ontology edge to specify that friends are also people.

The techniques introduced in [77], and their flavors, rely on the notion of source and target property cliques. They are defined using the concept of **source-related** and **target-related** properties, which are defined as follows:

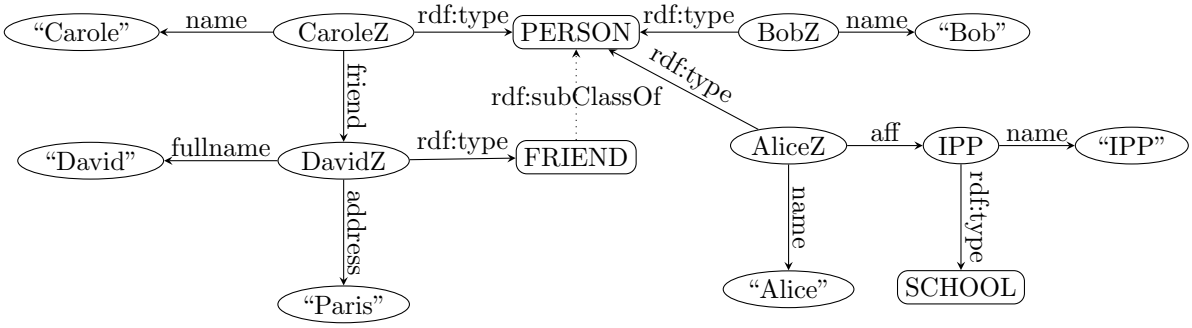


Figure 3.6: An RDF graph depicting four people, and one ontology edge saying that Friend is a sub-class of Person.

- Two RDF properties p_1, p_2 are *source-related* if there exist an RDF node s such that:
  - The node s is the subject of both p_1 and p_2;
  - Or s is the subject of p_1 and another RDF property p_3 such that p_2 and p_3 are source-related.
- Similarly, two RDF properties p_1, p_2 are *target-related* if there exist an RDF node o being the object of both p_1 and p_2; or o is the object of p_1 and another RDF property p_3 such that p_2 and p_3 are target-related.

A **source property clique** is the maximal set of RDF properties which are pairwise source-related. Similarly, a **target property clique** is the maximal set of RDF properties which are pairwise target-related. For instance, in Figure 3.6, the RDF properties `friend` and `name` are source-related because they both have a common parent, `CaroleZ`. Also, `aff` and `friend` are source-related, because `friend` and `name` are source-related, and `name` is source-related with `aff`. Therefore, one source property clique for Figure 3.6 is {`name`, `aff`, `friend`}.

All RDF quotient summaries described in [77] copy all schema triples from the input RDF graph to the quotient. Further, two flavors of summarization are introduced: one that first considers data triples (through property cliques), and then transcribes type statements from the input graph to the corresponding quotient nodes representing them; the other, on the contrary, first considers the type triples, and relies on property cliques only when types are absent.

Weak quotient summary

A weak quotient summary is a quotient of the RDF graph based on *weak equivalence*. Two data nodes are **weakly equivalent** if:

- They have the same non-empty source or non-empty target clique;
- Or they both have empty source and empty target cliques;
- Or they are both weakly equivalent to another RDF node in the graph.

Figure 3.7 depicts the weak summary of the RDF graph showed in Figure 3.6. Each rectangle corresponds to a summary node, each encompassing the summary node id and the nodes it rep-

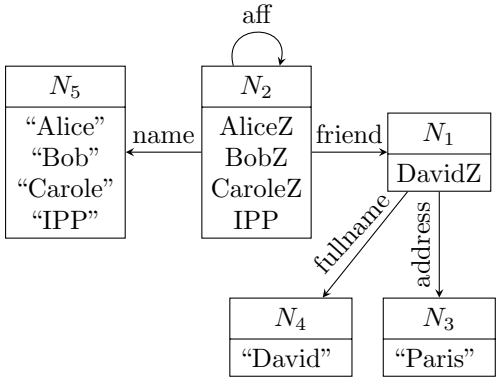


Figure 3.7: Weak summary.

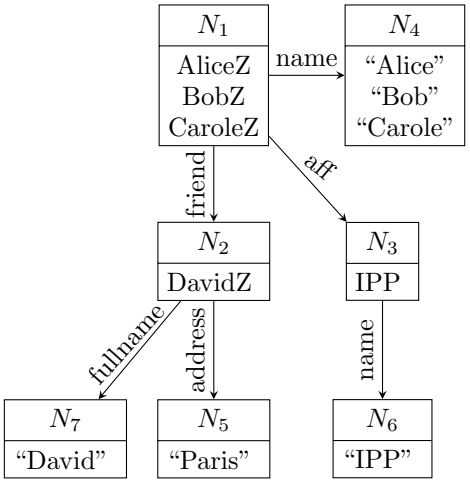


Figure 3.8: Strong summary.

resents. Alice, Bob, Carole, and IPP resources have been grouped together because they are all weakly equivalent due to their common outgoing property `name`. David is not included because it does not have the same source or target property clique as the others. Three summary nodes have been created out of the RDF values: N_3, which represents David's address, N_4, which represents David's name and N_5 containing all name values. Note the cycle on N_2, labeled `aff`; this is because the resources describing people and the school (`IPP`) are in the same summary node. Observe that weak summaries tend to create few groups, putting together many RDF nodes that would be better separated, e.g., the `IPP` school should not be mixed with people.

Strong quotient summary

Similarly, a strong quotient summary builds on a *strong equivalence*. Two data nodes are **strongly equivalent** if:

- They have the same non-empty source and non-empty target clique;
- Or they both have empty source and empty target cliques;
- Or they are both strongly equivalent to another RDF node in the graph.

The **strong summary** is the quotient graph computed from an RDF graph whose nodes have been grouped based on their strong equivalence. Figure 3.8 outlines the strong summary computed on the RDF graph of Figure 3.6. In this summary, people resources (Alice, Bob, Carole, and David) are grouped in two summary nodes: David is in N_2 while others are in N_1 because David does not share any property with other people. The school shares properties, e.g., `name`, with people resources, but it has an incoming edge `aff` that people do not have. Therefore, they are not strongly equivalent, thus are not in the same summary node. Unlike weak summaries, strong ones tend to create many groups, separating nodes that could belong to the same group. For instance, N_1 and N_2 could be merged, as they all represent people.

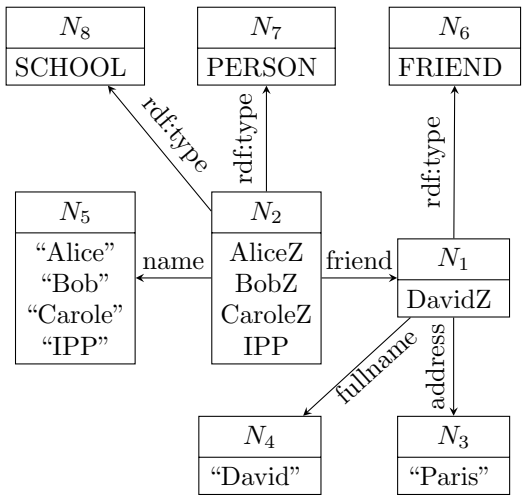


Figure 3.9: Data-then-type weak summary.

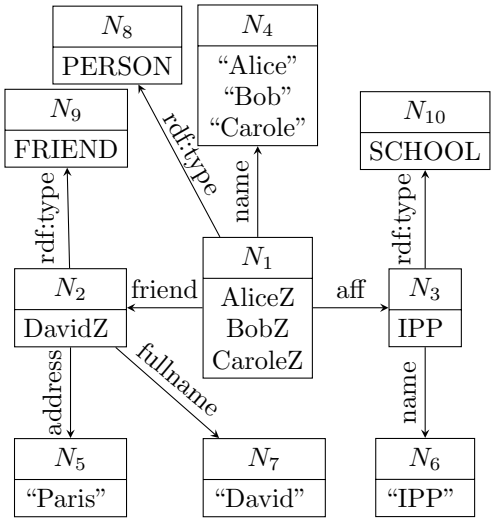


Figure 3.10: Data-then-type strong summary.

Data-then-type RDF quotient summary

The vanilla versions of the *weak* and *strong* quotient summaries only consider data triples (recall Section 2.6). However, type information is very precious when present. It reflects the human knowledge about resources and should be part of the summarization process. Therefore, the **data-then-type flavor** also considers type triples, i.e., the set of triples assigning an RDF type to a resource, as in ⟨`AliceZ`⟩⟨`rdf:type`⟩⟨`Person`⟩. A data-then-type (weak or strong) summary is computed as follows:

1. Summarize G using the weak, respectively strong, equivalence $\equiv$;
2. For each triple of the form n_1 `rdf:type` C, add an edge $m_1 \xrightarrow{\texttt{rdf:type}} C$ in the summary, where m_1 is the summary node representing n_1.

Figures 3.9 and 3.10 illustrate data-then-type weak and strong summaries. Observe that the vanilla weak and data-then-type weak summaries are identical, expect for the type information that is not present in the weak summary. This observation also holds for the vanilla strong and data-then-type strong summary. Those observations hold because data-then-type summarization techniques first summarize the graph (step 1) and then add type information on top of it (step 2).

Type-then-data RDF quotient summary

Another flavor is to consider type information as a first-class citizen and summarize first, based on RDF types, if they are available. When type information is attached to a graph, data producers have typically invested time and effort to model it; the outcome is also precious because data producers have the best knowledge of the data and its application domain.

For the type-then-data summary, two nodes $n_1, n_2 \in G$ are equivalent if:

- n_1, n_2 are both typed and have the same set of types;

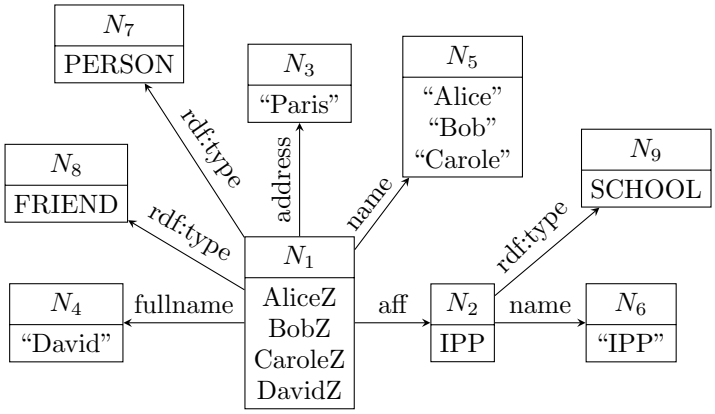


Figure 3.11: The type-then-data (weak and strong) quotient summary with type generalization.

- n_1, n_2 are both untyped and are equivalent according to the weak, respectively strong, equivalence $\equiv$.

All nodes in G are typed, thus the summarization respectively groups: the three `PERSON` resources, David, and the IPP school. Both weak and strong type-then-data summarizations lead to the same result; this is because all nodes are typed. Also, these two summaries coincide exactly with the data-then-type strong summarization, depicted in Figure 3.10. This does not hold in general, especially when there are more resources and more types.

Type generalization

When many types are attached to resources, type-first summarization may lead to (too) many equivalence classes. For instance, real world datasets about e-commerce transactions tend to use dozens of product types, e.g., Diary, Meat, Fish, Home Appliance, etc. The resulting summary computed with types includes each such type, thus scatters the resources into several summary nodes. Very often, when data types are numerous, a type hierarchy (DAG) comes with the RDF graph. In the case of the e-commerce graph, there could be a type Food, being a super-type for Meat and Fish; similarly, there could be a type Product, being a super-type of Diary, Food and Home Appliance.

For a given typed resource, type generalization uses the most general type associated to the resource type. For instance, a Meat resource would be summarized as if it was typed as a Product, the most general type associated to this type. Therefore, all Diary, Food and Home Appliance would be grouped in a single summary node. Similarly, users of different types, e.g., Seller and Buyer, would also be grouped if a super-type User exists.

Figure 3.11 shows the type-then-data summary obtained with type generalization. One can observe that people (of type `PERSON`) and Carole's friend (of type `FRIEND`) are grouped in the same summary node. This is thanks to the ontology triple saying that a `Friend` is a sub-class of a `Person`.

3.2.5 XML schema inference

One way to summarize XML data is to infer a DTD or an XSD schema (recall XML preliminaries in Section 2.4). Existing approaches include [74, 28, 27].

In [74], authors propose XTRACT, which, given an XML document: (*i*) generates a set of candidate regular expressions which, together, describe every XML path; (*ii*) *generalizes* the candidate expressions by replacing some patterns with regular expression meta-characters such as *; and (*iii*) applies the *MDL principle* (*Minimal Description Length*) which seeks to minimize the sum of the DTD length and the data instances length, to play both on the DTD conciseness and precision.

In [28], authors consider that the content of two XML elements with the same label does not depend on the full path to the root, but only on the i ancestors those elements may have, with $i \leq 3$ [124]. They define an XSD as being i-local if its model depends only on labels up to the i^{th} ancestor. For instance, recall Figure 3.2: there are four elements labeled `person`, the last one has a different set of ancestors, e.g., at depth $i = 3$. iXSD is an XSD generation tool which (*i*) applies iLocal, an algorithm which infers an i-local XSD from a corpus of XML documents; (*ii*) smoothens the obtained XSD by fusing types having the same structure; and (*iii*) also fuses types having *similar* structure, to obtain XSDs of reasonable complexity despite the sparsity and/or incompleteness of real-world datasets.

XML schema inference has also been implemented by companies, such as Microsoft, which provides an interface named *XmlSchemaInference* [188] to compute an XSD out of a (set of) XML document(s).

3.2.6 JSON schema inference

Despite the recent JSON schema proposals [170, 139], JSON documents are often shared and exchanged without a schema. However, to ensure that the application receiving JSON data knows what to expect, a schema could be very helpful, thus research works have proposed inferring JSON schemas. In [20], a parametric algorithm is proposed to summarize massive JSON datasets, which are viewed as collections of *records*. The approach is based on a *Map-Reduce* workflow. The *Map* phase assigns a type to each record value in the document. Next, the *Reduce* phase fuses records of similar type. In order to achieve good summarization results, authors provide two alternative methods for fusing types; we outline them below.

For instance, Figure 3.12 depicts a JSON collection of three maps, each representing a person. Note the common and different attributes that maps have, as well as their different depths.

Individual type inference

Given a JSON document, type inference builds a JSON document *of the exact same form* (respecting the initial nesting), but replaces any atomic value by its **inferred value type**. *Atomic values* comprise strings ($\mathbb{S}$ as defined in Chapter 2), numbers ($\mathbb{R}$), booleans (`true` and `false`) and null values.

Type inference applied on the JSON collection presented in Figure 3.12 leads to the JSON document in Figure 3.13.

```
[{
    "name": "Alice",
    "aff.": {"name": "IPP" }
  },{
    "name": "Bob",
    "aff.": {"name": null }
  },{
    "name": "Carole",
    "friends": [{
      "fullname": "David",
      "address": "Paris"
    }],
    "age": 26
  }]
```

Figure 3.12: A JSON document describing four people.

```
[{
    "name": STRING,
    "aff.": {"name": STRING }
  },{
    "name": STRING,
    "aff.": {"name": NULL }
  },{
    "name": STRING,
    "friends": [{
      "fullname": STRING,
      "address": STRING
    }],
    "age": NUMBER
  }]
```

Figure 3.13: The corresponding JSON document with inferred value types.

Fusion of similar records

Based on the typed JSON document, the fusion step seeks to *fuse* (*merge*) similar records, i.e., records with the same value type. The authors [20] propose two flavors of fusion. The first one is called **kind equivalence** and proceeds as follows:

- Atomic types are collapsed when they are identical; otherwise, they are combined using a union operator.
- For record types, i.e., maps or arrays, keys that are present in both records are collapsed (only one key is present in the final JSON document) and their non-atomic value types are recursively fused. Otherwise, when a key is present in only one record, the key is added, marked as optional and its value type is copied.

For instance, Figure 3.14 shows the schema of the JSON document in Figure 3.12 using the kind equivalence. The three map records are fused together because they have a common key, `name`. Further, their atomic value types are fused, e.g., `STRING` for people names, or unioned, e.g., `STRING + NULL` for affiliation names. The two keys that were not present in all records, i.e., `friends` and `age`, are added to the final schema, but marked as optional using the question mark.

The second option is to fuse records at a finer grain, using the **label equivalence**. In that case, atomic types are fused as with the kind equivalence. However, records are fused only if they have the same set of keys; other records are only unioned. For instance, Figure 3.15 illustrates the schema obtained with that fusion. The two first records are merged together because they have the same key set. Their atomic values are fused, e.g., for people names, or unioned, e.g., for affiliation names. The third map is not merged with the first two because it does not have the same key set, i.e., `friend` and `age` keys are not present in the first two records. Therefore, the label equivalence unions them, instead of merging them. This allows to preserve the precision at the expense of succinctness.

```
[{
   "name": STRING,
   "aff.": {
     "name": STRING + NULL
   },
   "friends": [{
     "fullname": STRING,
     "address": STRING
   }]?,
   "age": NUMBER?
 }]
```

Figure 3.14: The kind schema.

```
[{
   "name": STRING,
   "aff.": {
     "name": STRING + NULL
   }
 }+ {
   "name": STRING,
   "friends": [{
     "fullname": STRING,
     "address": STRING
   }],
   "age": NUMBER
 }]
```

Figure 3.15: The label schema.

3.2.7 Property graph schema inference

For property graphs, [33, 34] propose novel schema inference techniques, to summarize such graphs based on both node labels and properties. This improves authors' previous work [113], which was leveraging a less-performant Map-Reduce algorithm and which was able to consider only one aspect at a time, but not both. The Property Graph schema inference technique, denoted `GMM-Schema`, recursively builds a set of *base types*, i.e., *most general types*, which are further divided into sub-types based on their represented nodes' labels and properties. The sub-types then become the current base types and the process repeats until no further type can be divided into sub-types. For this, the algorithm leverages a *dividing hierarchical clustering* based on *Gaussian Mixture Models*. A Gaussian Mixture Model (GMM, in short) [127] is a probabilistic model assuming that a dataset can be viewed as a population and that sub-populations, i.e., groups of data points, follow a normal distribution.

Leveraging this, `GMM-Schema` goes as follows. First, the set L containing all the node labels is built. For instance, based on the property graph shown in Figure 3.16, $L = \{\texttt{PERSON}, \texttt{SCHOOL}, \texttt{FRIEND}\}$. Then, for each label $l \in L$, sorted by the decreasing order of their frequency:

1. Compute C, the set of PG nodes carrying (at least) the label l. When $l = \texttt{PERSON}$, we have $C = \{n_1, n_3, n_4, n_5\}$.
2. Construct a reference base type b_{ref} for C. A base type corresponds to the following triplet: $\langle l', k, E_b \rangle$, where l' is the set of labels nodes in C have (this is potentially larger than $\{l\}$ because a PG node may carry several labels), k is the set of most frequent properties C nodes have and E_b is the set of types b_{ref} extends. Intuitively, b_{ref} represents the most general type extended by all nodes in C, i.e., the parent type at the top of the type hierarchy. The reference base type for C would be $\langle \{\texttt{PERSON}, \texttt{FRIEND}\}, \{\texttt{name}, \texttt{address}\}, \emptyset \rangle$. Initially, E_b is empty because the reference does not extend any type.
3. Compute a feature vector d containing the similarity scores between b_{ref} and the base type of each C node. Intuitively, the more a C node has b_{ref} labels and b_{ref} properties, the better. The score is computed using the Dice coefficient, which, when applied to two sets, is a ratio between the number of common elements and the sum of the size of each set. For instance,

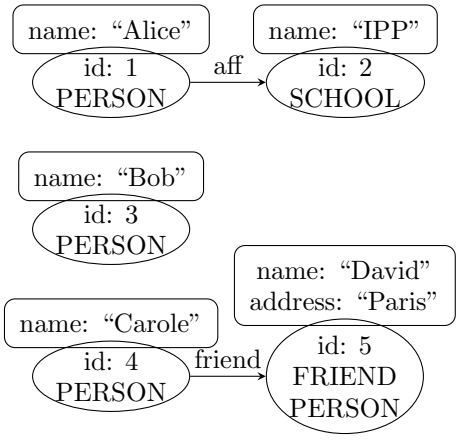


Figure 3.16: A Property Graph describing four people.

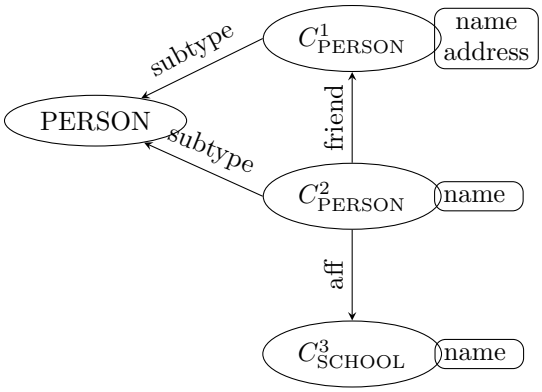


Figure 3.17: The Property Graph schema inferred from Figure 3.16.

for n_1, we have $k = \{\texttt{name}\}$ and for b_{ref} we have $k = \{\texttt{name}, \texttt{address}\}$, thus Dice coefficient for properties would be: $\frac{2*|\{\text{name}\}|}{1+2} = 0.66$. For types, we would also obtain 0.66. Then, these two scores are combined to get the final similarity score that is stored in the feature vector.

4. Fit a Gaussian Mixture Model with d and 2 normal distributions; in general, a GMM supports n normal distributions. This leads to a set of n mixture components, denoted θ_i. Each θ_i component corresponds to a sub-type of b_{ref}.

5. Using the n mixture components, classify C nodes in n sub-types, denoted C^i_l, based on how similar their base type is to b_{ref}. This is encoded in the feature vector d that has been given to the GMM. For instance, we obtain $C^1_{\texttt{PERSON}} = \{n_5\}$ and $C^2_{\texttt{PERSON}} = \{n_1, n_3, n_4\}$ because n_5 has the most similar reference type to b_{ref}, others are very similar to each other but are less similar to b_{ref}, thus belong to the second sub-type.

6. Apply hierarchical clustering on each sub-type C^i_l and stop when no subsequent sub-type is found. Finally, $C^1_{\texttt{PERSON}}$ and $C^2_{\texttt{PERSON}}$ are fed again in the algorithm but no more sub-type can be extracted, so the algorithm stops here for these two types and continues on the next label in L.

Finally, each sub-type becomes a schema node type, representing a unique combination of node labels and properties. The type hierarchy can be easily derived by adding an edge from a type to its n sub-types. Figure 3.17 shows the final schema obtained from the Property Graph presented in Figure 3.16. Observe that nodes of type `Person` are in two groups, both sub-types of the initial Property Graph node label `Person`.

3.2.8 Summarizing data of multiple models

So far, we discussed existing works for dataset summarization, and showed that they are tied to only one data model at a time. Following this idea, recent surveys [135, 137] expose three main reasons why we still lack a data model-independent approach:

1. *Data models* are very varied, ranging from tuples, to trees, graphs, etc. Quoting [137]: "most techniques are input data-model specific, [...] input datasets must all conform to a single data model, [...] and much of the evidence used to inform the schema inference process is specific to that data model. This has allowed existing approaches to exploit model-specific features to carry out inferences, but reduces applicability".
2. The summarization *output* is very varied from one data model to another, but also even for a given data model. For instance, JSON summarization techniques may output a tree, a JSON document or a schema with types. XML ones lead to XSD or DTD documents, trees or regular expressions. For RDF graphs, we may obtain a set of classes, a (summarized) RDF graph, or a table. For non-technical users, some of these outputs are hard to understand, in particular regular expressions or schema notations. On the contrary, novice users such as journalists are generally familiar with tabular data which they sometimes manipulate in spreadsheets; are better able to apprehend diagrams than text; and favor simplicity (or at least complexity that comes in a controlled fashion, only when users ask for it).
3. *Evaluation metrics* of the surveyed papers are primarily focused on the execution time and effectiveness (only). Effectiveness may be measured in different ways, but most pertinent measures rely on succinctness (*how much smaller the inferred schema is, compared to the data?*), precision (*how close is the inferred schema to a ground truth schema?*) or coverage (*how many data elements are represented in the inferred schema?*). According to [135, 137], scalability experiments are often lacking, due to the lack of community-accepted benchmarks, thus comparison with other approaches is difficult. A core reason for that is because of the diversity of data models on which summarization applies. Also, summarization code is available for less than half of the methods surveyed there.

As outlined above, no existing summarization method is capable of handling several, different data models, due to the above-mentioned challenges. In this thesis, we propose such a method, also aiming for intuitive summaries that any user can understand, regardless of their initial IT skills.

3.3 Structured and unstructured querying

To explore unknown datasets, users may query the underlying (integrated) data [97, 130]. Very often, users want to search for entities of interest and their connections. In our context, users query with only a partial idea of what they want to see. Mainly, they may have few particular keywords in mind (as in "how do H. Greim and Monsanto relate in my data?"). Also, they do not know in advance the complexity of the relationships they are looking for. For instance, companies usually do not directly grant researches of interest to them, they go through intermediary actors. Furthermore, users want to find entity connections regardless of their directionality; allowing forward and backward edge traversal considerably raises the problem complexity. For this querying task, users have two options: issue **structured queries** (Section 3.3.1) or use **keyword search** (Section 3.3.2). If the dataset is a **knowledge graph**, dedicated querying techniques have been devised (Section 3.3.3). We also review dataset search techniques (Section 3.3.4).

3.3.1 Structured querying

When users have an idea of what they are looking for and are familiar with the dataset structure, they can express **structured queries**. Such queries have the advantage to be very efficient, even

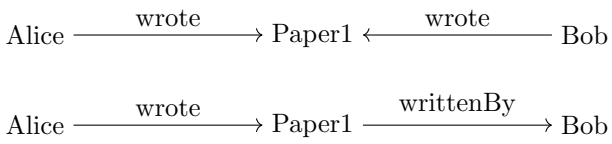


Figure 3.18: Bi-directional path (top) and uni-directional path (bottom) connecting `Alice` and `Bob` to their common publication, `Paper1`.

```
SELECT n1.label, n2.label, n3.label
FROM nodes n1, edges e1, nodes n2, edges e2, nodes n3
WHERE n1.label='Alice' AND n3.label='Bob'
  AND ((e1.s=n1.id AND e1.t=n2.id AND e2.s=n3.id AND e2.t=n2.id)
    OR (e1.s=n1.id AND e1.t=n2.id AND e2.s=n2.id AND e2.t=n3.id)
    OR (e1.s=n2.id AND e1.t=n1.id AND e2.s=n3.id AND e2.t=n2.id)
    OR (e1.s=n2.id AND e1.t=n1.id AND e2.s=n2.id AND e2.t=n3.id)
  );
```

Figure 3.19: The SPARQL query to ask for `Alice` and `Bob` connections.

on large data, and return the *complete* set of results. In the relational setting, users can write **SQL queries**. For graphs, **SPARQL** [173] or Cypher [167] are mostly used. When users do not know the length of the connection between two entities, they cannot use traditional structured queries mentioned above. Indeed, they need to specify in some way that their entities are connected *at a given distance*. One can specify such a query using **recursive queries** in SQL or **reachability queries** in the graph model. As of now, SPARQL and JEDI [5] check for paths between query variables, i.e., whether there exists at least one path connecting expected entities; it does not return the actual corresponding path(s). For instance, users may check whether a path of the form $x \xrightarrow{mother\_of*} y$ exists to know whether one staring on a node x can reach a node y by walking only on `mother_of` edges[1]. In contrast, a GPML query (GPML stands for *Graph Pattern Matching sub-Language* [53]) returns the expected paths. However, GPML is a standard in the making, i.e., at the time of this writing, no complete GPML query processing engine is available.

Structured queries require users to have at least some knowledge of the dataset schema; this is typically hard for users discovering datasets, or lacking IT skills, such as journalists. Also, the data may need to be traversed forward and backward, depending on how the data is modeled; this considerably increases the complexity and the query writing becomes cumbersome. For instance, Figure 3.18 shows that `Alice` and `Bob` have a common publication, namely `Paper1`; they may be connected using a uni-directional path (top) or a bi-directional path (bottom). To find how `Alice` and `Bob` relate, one needs to ask for all the possible connections if one has no prior knowledge on how the data is modeled. Figure 3.19 shows the corresponding SQL query: we ask for the four possible connections between `Alice` ($n1$) and `Bob` ($n3$) using an intermediate node $n2$, i.e., $n1 \rightarrow n2 \leftarrow n3$, $n1 \rightarrow n2 \rightarrow n3$, $n1 \leftarrow n2 \rightarrow n3$ and $n1 \leftarrow n2 \leftarrow n3$.

[1]The asterisk in the edge label indicates zero, one or an unlimited number of times, as in the regular expression vocabulary.

To facilitate querying for users lacking IT skills, numerous works have been conducted in many research areas, which we list here:

- **Interactive, incremental querying methods** [55, 154, 56, 61] build, with the feedback of the user, a query returning data they expect. For instance, [55, 154] leverages the *query-by-example* paradigm where users input a set of data examples in the system, which will further generalize the corresponding query and return the matching tuples. If users do not know which examples to input, [56] incrementally builds a user exploration profile based on few user-annotated examples, which the system uses to suggest tuples as relevant as possible. In the same direction, examplar queries [129] allow users to specify queries they are interested in, e.g., "Helmut Greim acknowledges Monsanto, which produces RoundUp", and obtain isomorphic results, i.e., those that have the same structure but not the same values, e.g., "Ursula von der Leyen acknowledges Pfizer, which produces Covid-19 vaccines".
- **Guided query writing approaches**, such as [61, 105], aim at helping users write syntactically and semantically correct queries to retrieve data. Such systems are also able to recommend attributes to use and tables to join.
- **SQL query generation**, such as [66], translates a set of keywords, e.g., "count author publications", into a SQL query. Some other works put a high-level language on top of the query engine. For instance, [110] provides Scala constructs to build queries, which will be further converted to SQL queries.
- **SQL query recommendation** systems model *user profiles* [58, 45] in order to recommend queries that might be of interest to them. To generate such recommendations, the system will rely on information gathered from the querying behavior of past users, as well as the queries posed by the current user so far.
- **NL2SQL** (Natural Language to SQL) methods, such those described in the survey [106], may also be used to gather the user query as a sentence and generate the corresponding SQL query using sets of rules [149, 152, 115] or deep learning methods, e.g., deep neural networks [151, 164, 178, 198]. Note that users still need to be aware of the database schema to be able to formulate a query.
- **Graphical query languages**, as in [107, 155], let users draw their query using "boxes", representing entities, and arrows for relationships. Because users are given the set of entities and relationships, they can compose their query without having a prior knowledge on the database. However, they still require to understand the conceptual model of entities and relationships (recall E-R model in Section 2.2).

All the above-mentioned techniques aim at making the query writing process more user-friendly. Some of them ask users to give examples of what they are looking for, others let them draw or express using natural language their query. Note that most of them require users to know the database schema in advance and/or understand how objects relate in the relational model.

3.3.2 Keyword-based search

Keyword-based search systems (KBS, in short), such as [6, 13], are designed for users who are not aware of the data structure but have few keywords of interest in mind. Those keywords will be used to identify data items relevant to the user, and (typically) how they are related to each other.

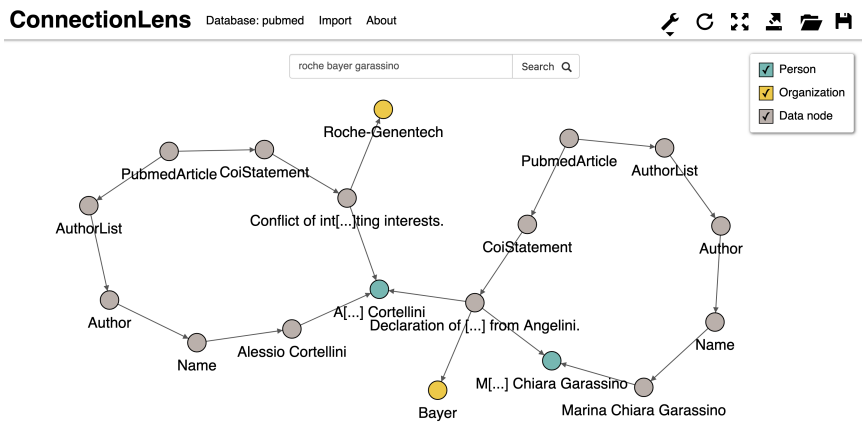


Figure 3.20: Keyword search query on the PubMed example presented in Figure 1.4 where users ask for connections between "Roche" (a Swiss multinational healthcare company), "Bayer" (a German multinational pharmaceutical and biotechnology company) and "Garassino" (an internationally recognized expert in the treatment of thoracic tumors).

Note that this kind of search does not allow formulating complex, very specific queries (as opposed to structured queries). Keyword search has been designed for relational databases [6, 29, 4, 93], XML data [70, 21, 120] and graphs [13, 62, 88]. Specifically, keyword-based search systems:

- On graph datasets, return trees (or sub-graphs) connecting nodes, such that each node matches one query keyword (see Figure 3.20);
- On tree datasets, return sub-trees connecting such nodes;
- On a relational database, return so-called Joined Tuple Networks, i.e., a set of tuples such that each of them joins with at least another via a natural join (primary key-foreign key join), and such that each keyword is matched by one attribute in one of the tuples.

Surveys on keyword search [196, 165, 197, 120] show that:

1. Keyword search *output* is often not exhaustive, but instead, is limited only to the answers that *best* match the keywords, according to a *score*. This avoids overwhelming the user with very large result sets, that may arise if the dataset is large and/or well connected.
2. Keyword search algorithms potentially face the *Group Steiner Tree Problem* (GSTP, in short), which makes keyword search a very costly operation. This problem (extending the Steiner Tree Problem) seeks, in a weighted graph, to connect several trees to a root node while maximizing an objective function, e.g., achieve the minimum total edge weight. This problem is known to be NP-hard [99], but many heuristics have been proposed, e.g., limit the number of results, enumerate smaller trees first to build larger answer trees on them, etc.
3. In the literature, many works [29, 4, 70, 21, 116] focus on *unidirectional search*, i.e., edges may be chained in a path only if the target of the first edge is the source of the second edge (recall Figure 3.18). However, proceeding in this way, many interesting results may be missed, depending on how the data is modeled, thus connected. The GAM algorithm [13] applies bi-

directional search to allow the algorithm to connect edges regardless of their directions. This considerably increases the search complexity, thus the algorithm has to be run with a time-out and will typically not explore its full search space.

4. The results are sorted according to a *scoring function*, which may be hard to specify or interpret for non-expert users. Also, keyword search algorithms use several parameters and heuristics, thus it is hard to understand why an answer is ranked better than another, especially for non-technical users. Combined with the fact that keyword search algorithms typically prune (do not enumerate) some of the results, this makes it hard for users to be convinced that the system returns all the results they would be interested in seeing.

Furthermore, journalists are also very interested in other connections they do not think of or see in the data. For such connections, they do not have keywords in mind and prefer to ask for all the connections for a given set of entity types (as in "how do people and companies relate in my data?"). To the best of our knowledge, prior to our work, there has been no method to automatically identify interesting paths in a graph, without asking the users to provide initial examples (which they may have a hard time doing).

3.3.3 Exploration of complex datasets

Knowledge graphs (KGs in short), such as the Google Knowledge Graph [185], Yago [138] or DBPedia [18], aim at representing all the knowledge of a given topic or of any topic (as an *encyclopedia*). It is often not possible to query them without prior knowledge and technical skills. Also, no summary can be both compact *and* comprehensive when computed on a knowledge graph due to their high conceptual complexity. To overcome those issues, several dedicated techniques, surveyed in [117], have been proposed to apprehend a knowledge graph the user is not familiar with, and we list them below:

- **Exploratory search** [118] allows users to dig into the data with a "tentative" query they will refine based on the results. For instance, one can inspect and query nodes around `Helmut Greim` until they have fulfilled their information needs about the toxicologist.
- **Exploratory analytics** [49, 96] compute statistics, and support multi-dimensional analytics as well as analytical queries. For instance, users may first ask about the general statistics of the knowledge graph, e.g., the number of entities per type and connectivity between types. This lifts the issue of not being able to build a compact *and* comprehensive summary of it. Next, they can ask for more topic-targeted queries, such as "*what are the top-10 toxicologists who reached an H-index about 50 in the last decade?*".
- Users can also look at the **knowledge graph constraints**, also known as *validating shapes* [143], to get a broad picture of what is in the graph. Such constraints express logical facts such as "*a professor works for a department*", "*a publication can have one or several authors*", etc. However, note that such constraints lack graphical representation and are hard to understand and exploit for non-expert users.

Above-mentioned approaches are dedicated techniques to help users explore graphs with a high conceptual complexity. They allow to get a broad overview of the data in the knowledge graph; next, users can leverage structured or unstructured queries to explore more precisely the data based on their needs and skills.

3.3.4 Dataset search

Users may also want to query data storehouses in order to find related datasets to the ones they already collected, e.g., to augment or improve them. This task is known as **dataset search**. The goal is to find, in a large repository, a set of relevant datasets given an information need, e.g., expressed through keywords [37, 41, 140], structured queries [68, 89], or similar tables [201]. Dataset search techniques often leverage word embeddings, as in TSBERT [47], and/or an ontology [89]. After finding complementary datasets, users can further summarize and query them to assess their relevance in their application context.

3.4 Summary

So far, we presented the different related works existing in the research areas of our interest, namely heterogeneous data integration, data summarization and data querying.

With respect to these areas, the positioning of the thesis is outlined below:

- For the *integration* part, our approach is based on a graph model, but stored in a relational database, in order to benefit from both their advantages: flexibility of the graph model; efficiency and optimization of the relational databases.
- Regarding the *data summarization*, the main novelty and strength of our approach is to apply to many data models (after an initial step that we argue must remain data model-specific). We leverage the dataset structure and content, to produce a user-oriented result; most of our decision are taken algorithmically and can thus be inspected and explained. Our solution does leverage trained language models for an entity classification tasks, to exploit the linguistic signal contained in attribute or edge labels.
- Our *query* approach does not assume any particular IT skills, such as writing queries, or user knowledge when specifying keywords, and it returns the *complete* set of paths connecting certain entity types. Also, it does so in and across datasets thanks to our graph integration of heterogeneous data sources.

More detailed elements of the positioning will be presented in the respective chapters, in order to be able to refer to individual techniques.

4

A unified view of semi-structured data formats: the graph representation

Chapter Outline

Chapter Abstract. In this chapter, we present how we convert data of heterogeneous data models into a unique, common paradigm: a directed graph. We first describe the conversion from relational (Section 2.3), XML (Section 2.4), JSON (Section 2.5), RDF (Section 2.6) and property graphs (Section 2.7) to a directed graph model (Section 4.1). Next, we explain how we use Entity Extraction tools to identify Named Entities, such as people, places, company names, in the data values (or leaves) of the data graph (Section 4.2). This can be seen as a form of graph enrichment. Finally, we discuss how to obtain, from the graph representation of heterogeneous data sources, a uniform directed graph (Section 4.3).

4.1 Target model: directed graphs

Directed graphs are a natural model to encode heterogeneous data, since they generalize all the other models. Recall Chapter 2: a graph G is a set of nodes N and a set of edges E, which may be both labeled, including with the empty label ϵ. When modeling different data models as a graph, we could choose either fine granularity data modeling, or coarse granularity. When fine granularity is used, nodes and edges have exactly one label, as is the case in the RDF standard. With coarse granularity, nodes may encapsulate their own attributes, in the style of property graphs. We chose the former for the following reasons:

- Fine-granularity nodes are more natural when representing RDF, JSON, or XML data;
- Conversion to coarse granularity would require turning some RDF, JSON, or XML nodes into coarse-granularity nodes, while other nodes of the initial dataset would become attributes of the coarse-granularity nodes. Instead, our goal is to make such judgments at a higher (conceptual) level (*which nodes are entities, and what are their attributes, possibly nested?*), based on a deeper analysis (see Chapter 6).

The transformation of any (set of) semi-structured dataset(s) into a fine-granularity graph, within the ConnectionLens platform, has been introduced in [13, 44]. In our setting, we call the graph computed by ConnectionLens out of one individual dataset the *data graph* G_0. Note that ConnectionLens assigns a simple integer ID to every node in G_0. In this chapter, we adopt the following convention when referring to nodes: N_i^a designates the data node whose identifier is i and carrying the label "a", while N_i designates the data node of id i (regardless its label).

For illustration, Figure 4.1 shows the data graph G_0 obtained from a dataset about conferences where authors have written (`hW`) some papers; each of which is written by (`wB`) that author. When papers are published in (`pIn`) a conference, authors may also be invited by (`inv`) the conference organizers. Each circle is a data node; each of which shows it `id` (the number in italic) and its label. To differentiate structural from $\mathscr{L}$-leaf nodes, we quote the latter. Each arrow between two nodes is a data edge, possibly annotated with its label (in italic). Red edges are those involved in cycles.

4.1.1 Relational data

We start by explaining how tabular data is converted into a graph.

A CSV file is converted as follows. Each tuple leads to an unlabeled node. Each tuple attribute leads to a leaf node, labeled with the respective attribute value. A directed edge leads from the tuple node to the attribute leaf node. If the CSV file has a header indicating attribute names, the edge is labeled with the respective attribute name; otherwise, the edge has an empty label.

A relational database is converted to a directed graph in a similar way: every tuple, and every attribute, lead to a node, with directed edges connecting them and labeled with the attribute name. Further, in the presence of a primary key-foreign key constraint of the form "$R.a$ is a foreign key referencing $S.b$", where a, b are sets of attributes from R, respectively, S, each node nr corresponding to a tuple $r \in R$ tuple has an outgoing edge labeled a pointing to the S tuple node ns created for the respective tuple $s \in S$. This modeling has been introduced in works focused on keyword search in relational databases, since [4]. When such primary key-foreign key pairs arise, the attribute nodes corresponding to $R.a$ and $S.b$ are removed (the edges between the tuple nodes replace them).

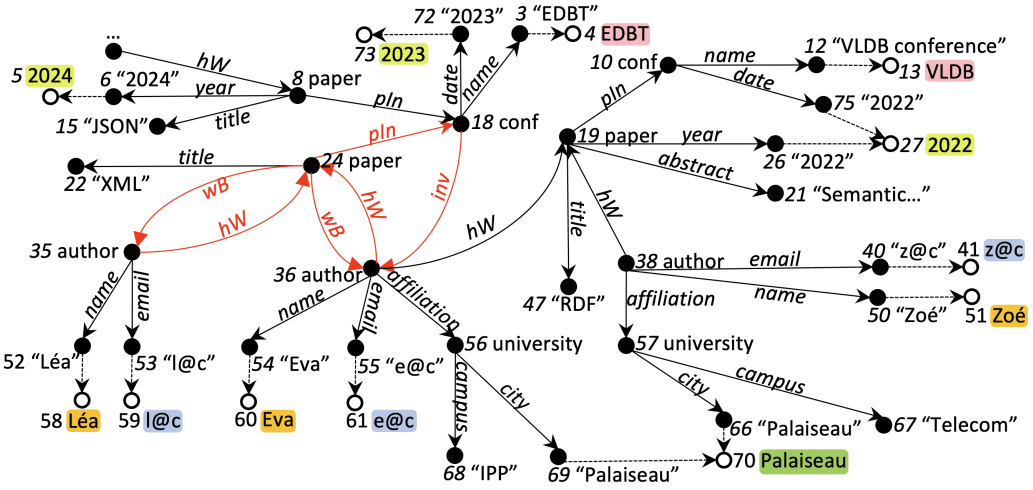


Figure 4.1: Example of a data graph.

```
1 <document>
2   <book authors="a1 a2">XML</book>
3   <authorList>
4     <author id="a1">Alice</author>
5     <author id="a2" city="Paris">Bob</author>
6     <author id="a3" city="Paris">Carole</author>
7   </authorList>
8 </document>
```

Figure 4.2: An XML snippet describing a book and its authors.

4.1.2 XML documents

XML documents lead to trees, thus are easily mapped to a graph representation. The document root node, all element nodes and all text nodes lead to data nodes in G_0. Such nodes may be connected by unlabeled edges, modeling the "parent-child" relationship. Next, for each attribute value, a node is created and is connected to its respective element node by a directed edge labeled with the element name, concatenated with the @ separator, concatenated with the attribute name. For instance, Lines 4 and 5 in Figure 4.2 lead to four nodes: $N_1^{authorList}$, N_2^{author}, $N_3^{\text{"a1"}}$, $N_4^{\text{"Alice"}}$. They are connected by two parent-child edges, respectively $N_1 \rightarrow N_2$ and $N_2 \rightarrow N_4$, and an element-attribute edge, respectively $N_2 \xrightarrow{author@id} N_3$. The obtained data graph is shown in Figure 4.3.

Because most of the XML datasets available online do not come with an XSD or a DTD document, we identify ID-IDREF connections (recall Section 2.4) by *profiling* the data. We only consider identifiers and references appearing in *attribute values*; this is a subset of all ID-IDREF cases according to the XSD formalism because they can be defined both on attributes and element values. Note that this restriction will significantly reduce the problem complexity without missing too much information, mainly because it is very frequent that ID-IDREF connections are set in XML attributes. We start by looking for candidate sets for identifiers and references. Given an

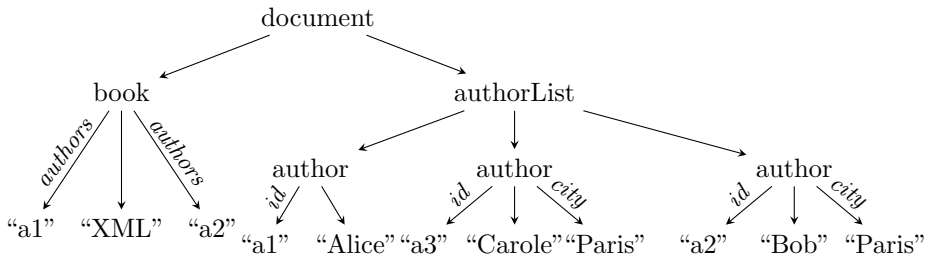


Figure 4.3: The data graph for the XML snippet in Figure 4.2.

XML element name e and an XML attribute name a, we denote by $S^{e@a}$ the set of data nodes created out of corresponding XML attribute values. For instance, in Figure 4.3, we have $^{\text{author@id}} = \{$"a1", "a2", "a3"$\}$. Based on $S^{e@a}$ sets, we compute the sets $I^{e@a}$ and $R^{e@a}$, respectively of ID and IDREF values, as follows. Any set $S^{e@a}$ having all its values distinct is also an $I^{e@a}$ because it comprises candidate ID values. Further, any set $S^{e@a}$ is also a $R^{e@a}$ set because any set of XML attribute values is a candidate set of references (IDREFs). For instance, in Figure 4.3, we have $I^{\text{author@id}}$, but not $I^{\text{author@city}}$ because it contains duplicates. A candidate set of references may also be a candidate set of identifiers: this happens when the set of references contains no duplicates. For two given sets $I^{e@a}$ and $R^{e'@a'}$, we consider the following assertions:

- When $R^{e'@a'}$ is a **strict subset** of $I^{e@a}$, $R^{e'@a'}$ makes a reference on the set $I^{e@a}$;
- When $R^{e'@a'}$ is <u>not</u> a **strict subset** of $I^{e@a}$, $R^{e'@a'}$ makes a reference on the set $I^{e@a}$ and conversely (no strong insight on the direction of the connection);
- When $R^{e'@a'} = I^{e@a}$ and the XML attribute a is expressly labeled `id` in the data, $R^{e'@a'}$ makes a reference on the set $I^{e@a}$.

Otherwise, the two sets have nothing in common, thus the pair is discarded. Next, for any set referencing another, each element labeled e' points to the element labeled e, whose value for the attribute a' equals the value of the attribute a (whose parent is e). For instance, from Figure 4.3, we obtain the data graph drawn in Figure 4.4. The `book` node points to two author nodes with two edges labeled `book@authors`. The value nodes initially created for the references, i.e., "a1" and "a2" `authors` values in Figure 4.2, are removed from the data graph as well as the edge connecting them to their parent. Once ID-IDREF connections have been identified, the data graph resulting from an XML document is no longer a tree, and it may actually have cycles.

4.1.3 JSON documents

We transform JSON documents into trees (particular cases of directed graphs) as follows:

- Each map, respectively array, leads to an unlabeled node;
- Each value leads to a node labeled with the string value;
- An edge labeled with the map key connects the map node to the map value (either an $\mathscr{L}$-leaf value or another element);

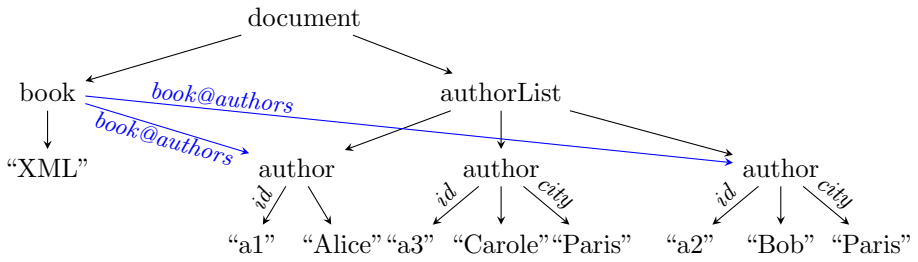


Figure 4.4: The data graph for the XML snippet in Figure 4.2 with ID-IDREF edges.

- An unlabeled edge connects each array node to its elements (either $\mathscr{L}$-leaf nodes or other elements).

Note that in principle one could profile JSON documents trying to find an equivalent of ID-IDREF links. We did not do this since it seems that such PK-FK encoding in JSON is less prevalent than in XML (where it has been specifically described both in the DTD and XSD standards).

4.1.4 RDF graphs

RDF graphs naturally lead to a graph representation. Each triple $\langle \texttt{s} \rangle \langle \texttt{p} \rangle \langle \texttt{o} \rangle$ is modeled as an edge labeled p, between the N_i^s and N_j^o, respectively carrying identifiers i, j and labels s, o. There is exactly one node (with identifier i) per RDF graph resource labeled with the URL s, and similarly, across the whole graph, for each literal, only one node is created (and automatically assigned an ID). This is in coherence with the semantics of the RDF data model (Section 2.6): each resource is unique, i.e., present only once in the RDF graph. Thus, each triple referring to an already existing RDF resource node will only increase the connectivity by adding an edge in the graph. Note that literal values used, e.g., for default values such as "true", "false" or "0", "1", may have a very high in-degree, if they are frequent in the data.

As stated in Section 2.6, we consider that RDF graphs we work with have been saturated, that is: they already contain all the schema or type triples that can be inferred from their triples using the possible (RDFS) ontology they contain.

4.1.5 Property graphs

Property Graphs (PG, in short) are ingested as follows:

- Each PG node leads to a node, assigned a unique identifier.
  - Each PG node label leads to a node labeled with that label and which is connected to its PG node by an edge labeled with the standardized RDF URI `rdf:type`. We do this for two reasons: (*i*) a PG node label cannot simply become a node label, because a PG node may have several labels, while in our model we assign a single label to each node; and (*ii*) as we will discuss in Chapter 5, we summarize Property Graphs using the same technique as for RDF graphs. From this viewpoint (and also taking into account how labels are used when creating Property Graphs), PG node labels are semantically close to RDF types.

  - Each PG node attribute leads to a node labeled with the attribute value; an edge labeled with the attribute name connects the PG node to its PG attribute value node.

- Each PG edge *without* proper attributes connecting two PG nodes leads to an edge labeled with the PG edge label;
- Each PG edge *with* proper attributes is converted into a node N_j (a fresh identifier) labeled with the edge label *el*. Two edges, respectively labeled *el*_`subject` and *el*_`object`, connect the edge source node to N_j and N_j to the target node. Each N_j attribute is converted as PG node attributes and attached to N_j.

4.2 Extraction of Named Entities

In Information Extraction (IE), a **Named Entity** (NE) is a real-world object, such as a Person, Place, Organization, Product, Event, etc., that can be denoted with a proper name. Such Named Entities can be found in text, using more or less advanced techniques; their family is referred to as Named Entity Recognition (NER). Technically speaking, NE extractors take as input a textual string and yield a set of triples of the form $\langle NE, \tau, c\rangle$ where NE is the identified NE label, τ is the type associated to that entity and c is the confidence the extractor has in NE and τ. Currently, the set of extracted entity types, denoted $\mathcal{T}$, includes Person, Location (places), Organization, Date, Email, URI, Hashtag and Twitter mention NEs; they can currently be extracted from English and French. We have at hand two kinds of extractors, of each which has different capabilities in terms of NER:

- Pattern-based extractors may extract Date, Hashtag, URI, Email and Twitter mention NEs;
- Extractors based on pre-trained Language Models identify Person, Location and Organization NEs.

Pattern-based extractors are internally composed of regular expressions. They are easily defined for dates, hashtags, URIs, emails and Twitter mentions, because of the simplicity of the pattern to recognize. Pattern-based extraction is extremely fast. For hashtags, URIs, and emails, it is error-free; for dates, it may introduce both false positives, e.g., consider 2023 a year when it is a page number, and false negatives, e.g., miss "Christmas Eve this year". The latter would be detected by a state-of-the-art temporal extractor such as HeidelTime [159], on top of which an extractor has been developed as part of this thesis. However, HeidelTime is very slow, thus in this work we extracted dates using patterns.

People, Organization and Location names can take multiple forms. Thus, ConnectionLens [13] extracts them using two extractors, each of them is based on a **pre-trained language model**. The first one is based on the **Stanford NER** [69, 122], while the second one is based on the Deep Learning **Flair** NLP framework [8][1], pre-trained on the CoNLL-2003[2] news articles dataset.

Named Entities are extracted from each $\mathscr{L}$-leaf node of the data graph (RDF or PG literal, XML text node or attribute value, or JSON value). Each such node t is sent to the extraction module, which performs the following steps:

[1] https://github.com/flairNLP/flair
[2] https://www.clips.uantwerpen.be/conll2003/ner/

1. First, all pattern-based extractors are applied on t, leading to a (potentially empty) set of NEs, denoted NE_t. Then, all tokens corresponding to NE_t are removed from t, to avoid re-analyzing these tokens by trained models (Flair or ChatGPT), which takes time and/or incurs financial costs. For instance, if t is initially "E. Macron and J. Biden met during the NATO summit in 2023 @nato", "2023" is extracted as a Date by the date pattern and "@nato" by the Twitter mention extractor. Then, t becomes "E. Macron and J. Biden met during the NATO summit in " (note the absence of "2023" of "@nato" at the end of the string).
2. Next, the trained-model extractor is called on the (remaining) t.
3. Each extracted NE is materialized by a new node in the graph, connected via an extraction edge (dashed arrows in Figure 4.1) to each $\mathscr{L}$-leaf node from which it has been extracted, e.g., as the "Palaiseau" place (green background). People are highlighted in orange, Emails in blue, Dates in yellow, Organizations in pink and Locations in green.

4.3 Graph normalization

The data graph G_0 obtained from an original dataset may exhibit some heterogeneity, in particular for what concerns its edges. If G_0 is obtained from an RDF graph, all data edges are labeled. If G_0 is built from an XML graph, edges corresponding to parent-child element pairs are unlabeled, but edges connecting elements with their attributes are labeled. When G_0 is built from JSON, most edges are unlabeled; for CSV files, the answer depends on the presence of a header.

For uniformity, we transform G_0 into an **unlabeled directed graph** $G = (N, E)$, where each labeled edge from G_0 has been transformed into a node. In details, any edge of the form $N_i \xrightarrow{l} N_j$ is replaced by a node N_k^l and two unlabeled edges $N_i \rightarrow N_k^l$ and $N_k^l \rightarrow N_j$ in G. Figure 4.5 shows the normalized graph obtained from the data graph in Figure 4.1.

The space complexity of graph normalization is bounded as follows: $|N_0| + |E_0| \leq |N| + |E| \leq |N_0| + |E_0| * 3$. This is linear in the number of elements in G_0 (nodes and edges).

4.4 Summary

In this chapter, we have shown how to convert several datasets, possibly originating from different data models, into a unique data graph. Beyond the version originally published in [13]; we extended it by creating an unlabeled data graph, which is necessary toward designing generic algorithms for semi-structured datasets. Moreover, Named Entity extraction is applied on the data graph values; this allows to better interconnect datasets based on their common entities/actors.

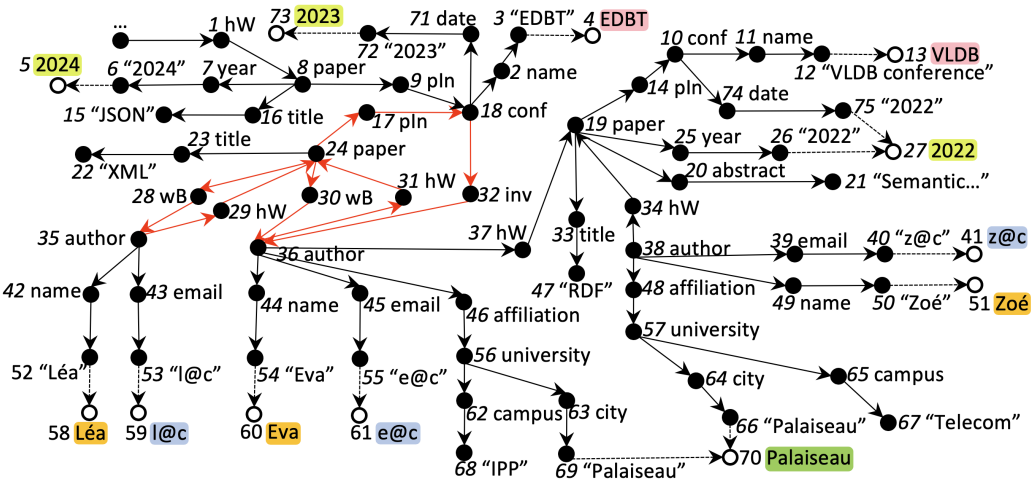


Figure 4.5: The normalized data graph obtained from the initial data graph in Figure 4.1.

5

From a data graph to a collection graph

Chapter Outline

Chapter Abstract. This chapter presents how we build a **collection graph**, a core structure for determining, in a given dataset, the main entities and their relationships (see Chapter 6); also, the collection graph will be used to enumerate paths between entities of interest (see Chapter 7). Informally, a collection is a set of data nodes that represent *data objects (or records) of the same kind*, regardless of the data model in which the dataset was structured. Toward identifying collections, we start by an analysis of how data creators encode information about records, their attributes, and relationships (Section 5.1). Next, we explain how the *kind* is encoded by each data model (Section 5.2). Further, we detail how the collection graph is constructed from the data graph, and based on the above analysis (Section 5.3). Finally, we show how to build a collection graph out of data graphs where several datasets co-exist (Section 5.4).

5.1 From applications to datasets: a unified perspective

We consider the data models discussed in Chapter 2, namely: relational, CSV, XML documents, JSON documents, RDF graphs, and Property Graphs. Toward abstracting datasets, we first analyze how each model describes a set of core features of any real-world application: application objects (at various granularities), data types, relationships, and information about application objects "of the same kind". We use "kind" (not "type"), because "type" has precise (but different!) meanings in different data models. In this work, "same-kind" designates *similarity from the application perspective*.

5.1.1 Records and values

We denote by **record** a piece of data in a dataset, describing an application domain object. Specifically:

- In a relational database, each tuple or attribute value is a record;
- In a JSON or XML document, each node is a record;
- In a PG or an RDF graph, each node is a record too.

Clearly, some records may *contain* other records. For instance, tuple attribute value records are contained in (or part of) the respective tuple record[1]. We call **leaf record** (or **value**) a record that does not (syntactically) contain any other records: relational attribute values; leaf nodes in JSON and XML documents; literals in RDF graphs; atomic values of node or edge attributes in a PG.

Each data model supports some **value types** such as String, Float, etc., described, e.g., in the ISO/IEC 9075:2011 standard for relational databases, or the W3C's standard [177] for XML and RDF. Even if declared of type String, values such as "Paul Jones", "paul@outdoor.com" or "https://twitter.com/pjones" denote, more expressively: a person name, respectively, an email, and a URI. Named Entities may be *inferred from the data*, through profiling [1], pattern matching or Named Entity Recognition (recall Section 4.2). Both our abstraction and entity path enumeration methods leverage them. For abstraction, it allows us to semantically classify the main entities (Section 6.8). For entity path enumeration, it connects Named Entities of interest in the data graph (Section 7.3).

5.1.2 Relationships

Next, we consider where and how **relationships** between application objects are expressed in the data.

A first category of relationships comprises *anonymous part-of* relations, e.g., the records of the maps within a JSON array are part of the record corresponding to the array; similarly, an XML element is part of its parent.

Next, we identify *named binary relationships*, which hold between:

- A tuple record and its value, the relationship is named with the relation attribute;

[1]This raises the question: where does one record end and where does another record start? This question is complex for some data models, such as XML, RDF etc. We will address it in Chapter 6.

- An XML element record and each of its attribute values - the relationship name is the attribute label;
- Two XML elements when one refers to another, e.g., when specified through schema information using a DTD or an XSD (recall Section 2.4); the relationship is named with the referenced node element name and attribute name;
- A JSON map record and its children: in this case, the relationship name is the attribute name (or key);
- A Property Graph node and each of its attribute values; the relationship name is the attribute label;
- A Property Graph edge is a named binary relationship; it may have its own attributes: the relational model also allows this, e.g., in a relation `WorksFor(personID, companyID, startDate)`;
- Each RDF triple naturally encodes a named binary relationship.

For instance, in the relation `Person(id, name)`, in the tuple `(1, 'Alice')`, a binary relationship labeled `name` holds between the tuple record and the value `'Alice'`. Similarly, in the XML element ⟨`person id='1' name='Alice'`/⟩, a relationship labeled `name` holds between the `person` node and the value `'Alice'`. Moreover, if a DTD or XSD states that an attribute `id` is a primary key (#ID) for ⟨`person`⟩ elements, and that `parent` is a foreign key (#IDREF) on person ids, then ⟨`student name='Bob' parent='1'`/⟩ leads to a binary relationship between the student and the person records. This relationship would be labeled `student@parent`.

Datasets may also encode relationships of a higher arity n, where $n \geq 2$. These are more rare, and we do not consider them in this work.

5.1.3 Same-kind records

Our last question is: how do data creators signal, in a dataset, **same-kind records**, that is, records describing real-world objects of the same (or similar) nature? In a relational database, all tuple records of the same table are of the same kind; the same holds for XML elements with the same name in a document. If an optional DTD or XSD schema is available, each element is assigned a type, and all same-type elements are of the same kind. Moreover, leaf records (values) participating in a given named binary relation, with non-leaf records of the same kind, are of the same kind. For instance, values of "name" attribute in the `Person` relation above, "name" attributes of ⟨person⟩ XML elements.

RDF and property graphs allow attaching to node records *zero, one or more RDF types, respectively PG labels*, e.g., a node can be a "FrenchCitizen", a "Student" and also a "PhDStudent". If an (optional) ontology is attached to an RDF graph, it may lead to *infer* some node types, e.g., if x is an UndergraduateStudent, then x is also a Student. In this work, we assume that the set of facts inferred from the ontology is *finite, already computed* [78], and *part of the RDF graph*, thus, we only consider saturated RDF graphs.

Some records carry *no explicit kind information*. This is the case for RDF and property graph nodes without types (or labels), non-leaf JSON records, and nodes created out of a CSV relation with no header. Instead, their kind is *implicitly* encoded in the data structure.

We call **kind name(s)** of a record:

- for a tuple $r \in R$, the relation name R;
- XML element names;
- RDF node types and Property Graph node and edge labels (when present).

Other records do not have kind names.

In semi-structured data models (JSON, XML, RDF, Property Graphs), records may lack an explicit type (label), or on the contrary, several explicit types (or labels) may be present for a given record. Thus, we need a method for deciding which records are of the same kind. We will describe such methods in Section 5.2.

5.2 Node equivalence in different data models

Introduced in Section 5.1, the *kind* information allows us to identify groups of nodes based on their *similarity from the application perspective*. Kind information is encoded *explicitly* in some data models, while in others, it is *implicitly* encoded. We view "being of the same kind" as an **equivalence relation**, and leverage it as introduced in Section 3.2.1, to group nodes that describe things/objects that are similar in the real world. Thus, two principles guide our partitioning:

(★) Whenever "kind" information is *explicit*, we should leverage it, as it reflects the dataset producers' application-domain knowledge.

(■) The number of equivalence classes should remain "reasonable", e.g., at most in the hundreds (as opposed to thousands or more), in order to produce meaningful and intelligible results meant for users. This is because abstractions and path exploration are meant for novice users, which should not be overwhelmed with details.

It is important to note that no "one-size-fit-all" equivalence relation can be used for all formats, thus **the partitioning is model-dependent.** Indeed, "kind" information is not always available in the same form, thus requiring different specifications depending on the data model. When several kinds are present, we need to decide which one prevails. On the contrary, when the kind is not available at all, we need to devise model-specific methods to group nodes.

Once a collection graph has been built, the next processing steps involved in abstraction and path exploration are based on it (see Section 5.3), thus are data-model independent.

5.2.1 Relational data

Relational data explicitly assigns only one "kind" to their records, which correspond to the tuples in the relational setting. Indeed, each tuple belongs to exactly one relation.

Let $R(a_1, ..., a_n)$ be a relation (coming from a relational table or a CSV file):

- The G nodes created from any tuple $r \in R$ are equivalent to each other, thus all belong to the same equivalence class.
- Edges in G_0 were labeled with the attribute names if they exist and have been normalized into nodes in G.

- All G nodes corresponding to an attribute a_i, and connecting a tuple node to its a_i value, are equivalent. They are grouped in an equivalence class labeled with the attribute name, if it exists.

- Next, the nodes created from attribute values $r.a_i$, for $r \in R$ and an attribute a_i, are equivalent. Their equivalence class is labeled with the attribute name and the suffix `#val` as in a_i`#val`.

- Finally, an equivalence class edge connects two equivalence classes EC_i and EC_j whenever there exist two G nodes N_1 and N_2 such that $N_1 \rightarrow N_2$ and N_1 is represented in EC_i, respectively N_2 is represented in EC_j.

This partitioning is the most natural: tuples of the same relation are equivalent, respectively, same-name attribute values are equivalent, etc. The partitioning can also be seen as a particular case of the path summarization described in Section 3.2.3. Indeed, tuples of a given relation, thus occurring on the same path, are grouped in the same equivalence class.

5.2.2 XML documents

XML trees also explicitly assign only one "kind" to their records. We leverage the label partitioning described in Section 3.2.2 and group nodes as follows:

- For what concerns XML elements, if a schema, such as a DTD or an XSD, is available with the data, we leverage it to respect principle (★) and group nodes of the same *type* in the same equivalence class. If not, XML elements with the same label are said equivalent and grouped in the same class. For instance, the `author` elements in Figure 4.2 leads to a class (representing the three `author` nodes). Another class will group all the three author names nodes.

- With respect to element attributes, recall that each G_0 labeled edge created for any attribute to connect the element node to its attribute value has been transformed, during the normalization, into a node containing the attribute name. As shown in the normalized data graph in Figure 4.5, the edge connecting the author node N_{35} to its name ("Léa") in Figure 4.1 has become the node N_{42}^{name}.
  We group such nodes as follows. For a given element name, e.g., `author`, and a given attribute, e.g., `name`, an equivalence class of the form `elemName@attrName` is created. For instance, in Figure 4.5, the equivalence class `author@name` will represent the nodes N_{42}, N_{44} and N_{49} but not N_2 and N_{11} as those attributes come from a `conference` element, not from an `author` one.

- Next, we turn to XML values. For each equivalence class of XML element values, we create an equivalence class of the form `elemName#val`. Respectively, for each equivalence class of XML attribute values, we create an equivalence class of the form `elemName@attrName#val`. For instance, the equivalence class `author@name#val` represents the nodes N_{50}, N_{52} and N_{54}, corresponding to the three authors' names.

- Finally, edges connect equivalence classes: for two equivalence classes EC_i and EC_j, there exists an equivalence class edge $EC_i \rightarrow EC_j$ whenever there exists a data edge for two G nodes N_1, N_2 such that $N_1 \rightarrow N_2$.

```
[{"name": "Bob", "aff": "IPP",
  "friends": [
    {"name": "Alice", "aff": {"lab": "Inria", "city": "Paris" }},
    {"name": "Zoe", "email": "zoe@company.com" }
  ]
}]
```

Figure 5.1: A JSON snippet.

5.2.3 JSON documents

For JSON documents, no explicit kind is defined and many nodes have empty labels. Therefore, a kind based on the node names, as for XML, is not a good indicator of the kind: all empty-labeled nodes would be grouped together in a single class. Instead, we consider equivalent nodes on the same path, following the strong DataGuide summarization (described in Section 3.2.3). We proceed as follows:

- Array and map nodes are grouped when they appear on the same path from the root. For instance, in Figure 5.2, an equivalence class contains the root array in Figure 5.1, another one contains the outer map, a third one contains the two maps representing Bob's friends, etc.
- Next, each JSON map key has been converted to a node while normalizing G_0 into G. Therefore, all the nodes for a given key on a given path from the root are grouped in an equivalence class. For instance, in Figure 5.2, nodes created out of Bob's friends `name`-labeled edges are grouped in the same equivalence class; so it is for the outer `name`, the outer `aff`, `friends`, Bob's friends `email` keys, etc.
- Further, we need to group JSON values. For each equivalence class of a given array, respectively a given map and a given key, all values nodes encountered at that place are grouped in an equivalence class. For instance, the equivalence class computed out of the friends' names, namely `name#val`, represents "Alice" and "Zoe".
- Last but not least, we add equivalence class edges as follows: whenever there exists a data edge $N_1 \rightarrow N_2$ for two G nodes N_1 and N_2, there exists an equivalence class edge $EC_i \rightarrow EC_j$ such that N_1 is represented by EC_i, respectively N_2 is represented by EC_2.

Using the partitioning described above, we observe that map nodes representing similar things may be separated in different equivalence classes if they are located in different places in the JSON document (thus, are not on the same path from the root). This is, e.g., the case with equivalence classes on the paths `[]`.ϵ.{} and `[]`.ϵ.{}.ϵ.`friends`.ϵ.`[]`.ϵ.{} representing the outer map, respectively the two friends maps. To overcome this limitation and to follow principle (■), we apply an extra summarization step on the partition computed out of JSON documents. Specifically, we fuse equivalence classes of map nodes having the same *property clique*, following the intuition they should belong to the same equivalence class (even if they do not occur on the same path). We proceed as follows.

First, we compute source property cliques on G_0 map keys, just like the Typed Strong summary does with RDF properties (recall Section 3.2.4). We denote by EC^{SC_i} the set of equivalence classes having the (source) property clique SC_i. For instance, in Figure 5.2, we obtain two source property

cliques, i.e., $SC_1 = \{$ name, aff, friends, email $\}$ for the outer and friends maps, respectively $SC_2 = \{$ lab, city $\}$ for the affiliation map. Then, we have $EC^{SC_1} = \{EC_1, EC_8\}$ and $EC^{SC_2} = \{EC_{12}\}$.

Next, for each property clique SC_i and its associated set of equivalence classes EC^{SC_i}:

1. We identify EC^*, the equivalence class in EC^{SC_i} having the lowest id. In Figure 5.2, this would be EC_1 for SC_1 and EC_{12} for SC_2.
2. Next, we fuse in EC^* all equivalence classes in EC^{SC_i}, meaning that all (map) data nodes represented by any $EC \in EC^{SC_i}$ are now represented by EC^*. This fusion is depicted in Figure 5.3: both the outer and friends maps are now represented in the outer equivalence class.
3. Recall that normalization introduced nodes that correspond to map edge labels in the original JSON dataset. Given two equivalence classes EC_x and EC_y, such that (i) $EC_x \rightarrow EC_y$, (ii) $EC_x \in \{EC^{SC_i}\} \setminus \{EC^*\}$ and (iii) EC_y represents k-labeled nodes (thus, k is a map edge label), several cases may arise:
   - If EC^* already has a child equivalence class EC_a, i.e., $EC^* \rightarrow EC_a$, representing k-labeled nodes, EC_y data nodes are "moved" to, i.e., represented by, EC_a; this is, e.g., the case of EC_9, which is fused in EC_2 (Figure 5.3);
   - Otherwise, EC_y is attached as a child of EC^*, thus $EC^* \rightarrow EC_y$; this is, e.g., the case of EC_{17}.
4. Next, we need to proceed carefully with equivalence classes of values, which may contain atomic values, maps or arrays. Reusing the notations EC^*, EC_a, EC_x and EC_y from above and considering a third equivalence class EC_z, such that (i) $EC_y \rightarrow EC_z$ and (ii) EC_z represents key values, several cases may arise:
   - If EC^* already has a grandchild equivalence class EC_b, i.e., $EC^* \rightarrow EC_a \rightarrow EC_b$, and both EC_b and EC_z contain *atomic values* (their label ends with `#val`). In this case, we fuse EC_b and EC_z.
   - Similarly, if EC_b and EC_z both contain *arrays*, we fuse them. The (child) equivalence classes containing array values are fused depending on the values they contain, i.e., strings, arrays or maps.
   - Similarly, if EC_b and EC_z both contain *maps*, we do nothing as this case is covered in step (2).
   - Otherwise, when EC_b and EC_z contain elements of different types, e.g., atomic values and arrays, we add EC_z as a new child equivalence class of EC_a.
5. Finally, we delete and/or change edges between the equivalence classes, to reflect the possible fusions applied above. We proceed as follows:
   - Each equivalence class edge $EC_x \rightarrow EC_y$, such that both EC_x and EC_y have been fused, is deleted.
   - Next, each equivalence edge $EC_x \rightarrow EC_y$, such that only EC_x or EC_y has been fused, is updated to point to the equivalence class in which EC_x, respectively EC_y, has been fused.

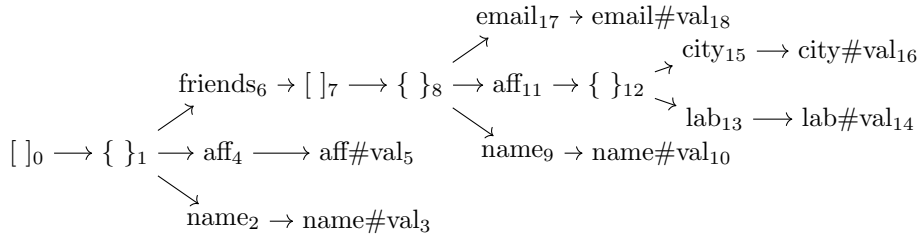


Figure 5.2: A visual representation of the partition after path-based summarization is applied.

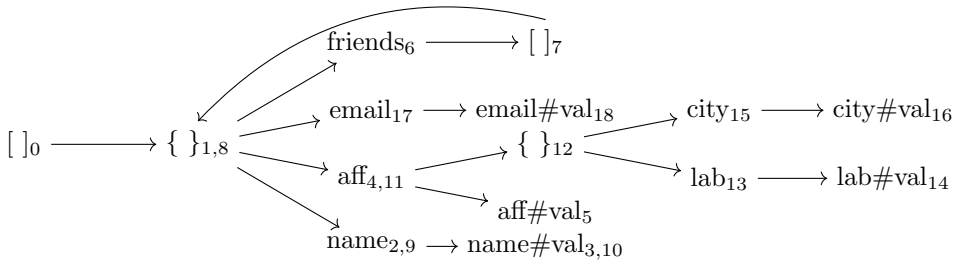


Figure 5.3: A visual representation of the partition after the clique-based further summarization is applied.

For instance, in Figure 5.3, EC_7 will now point to $EC_{1,8}$, the edge connecting EC_8 to EC_9, respectively EC_9 to EC_{10} are removed, etc.

The fusion of equivalence classes using property cliques described above bears some similarities with the construction of a minimal DataGuide (recall Section 3.2.3), in the sense that in both cases, we reduce the number of summary nodes. However, there are differences among the two. The clique-based approach only reduces the number of equivalence classes containing map nodes, while a minimal DataGuide reduces the number of equivalence classes overall. Also, the summary resulting from path-based summarization followed by clique-based factorization is not necessarily minimal: more equivalence classes could be fused using a state minimization algorithm (recall Section 3.2.3). In our example, this is the case of EC_0 and EC_7.

5.2.4 RDF graphs

RDF graphs attach zero, one or several explicit kinds to each record, which exactly corresponds to an RDF node. In such graphs, kind is expressed through *types*, as defined in the W3C standard.

To group together RDF nodes of the same kind, we rely on the *typed-strong* RDF graph summary, outlined in Section 3.2.4. Recall that resources having one or more types are grouped together by this summarization technique, if and only if they have the same set of most general types; this respects our principle (★). Next, typed strong summarization groups untyped nodes together based on the *cliques* into which their source and, respectively, target properties are grouped. These choices have been shown [77] to lead to a moderate number of equivalence classes, thus also satisfying principle (■).

For instance, the three paper nodes of the initial data graph G in Figure 4.1 have the same *source property clique*, defined as $\{pIn, title, year, wB, abstract\}$. Similarly, they all have the same target property clique: $\{hW\}$. Recall that, to obtain a robust partitioning method, two untyped nodes are considered equivalent if they have the same source and target property cliques. In Figure 4.1, the three paper nodes N_8, N_{19}, N_{24} have the same source and target property cliques, thus belong to the same equivalence class. Note that untyped nodes encompass resources, i.e., URIs, and literals: RDF literals are always grouped depending on their target property clique (as their source property clique is always empty: an RDF literal is a leaf in the graph).

The clique-based summarization technique operates on the initial data graph, not the normalized one, in order to leverage edge labels. Therefore, it only partitions the nodes in G that are also present in G_0, the original graph (prior to normalization). To also reflect the G nodes introduced by normalization, we proceed as follows. Consider an edge $EC_i \xrightarrow{p} EC_j$ connecting the equivalence class EC_i to another class EC_j using a p-labeled edge in the summary of G_0. For each such edge, a new equivalence class EC_k is created and represents p-labeled nodes that have been introduced while normalizing G_0 in G. This allows to represent all G nodes in the final partition. For instance, the newly created equivalence class EC_k would represent nodes N_{16}, N_{23} and N_{33} and connect the `paper` equivalence class to the one representing paper title values.

Finally, recall (Section 2.6) that RDF graphs may comprise also ontology (or schema) triples. Following typed-strong summarization (and more generally, the quotient summarization principles from Section 3.2.4), ontology triples are not altered in any way when constructing summaries: they are copied from the graph in the summary, as they are. This fits well with our goal to *summarize data*, while being guided by the ontology if one exists.

5.2.5 Property graphs

Property graphs attach zero, one or several explicit labels to their nodes; zero or one explicit labels can be assigned to an edge. Recall that property graphs model kinds using node, respectively, edge labels. As described in Section 4.1.5, we convert property graphs into RDF-like graphs: nodes have one label but zero, one or several (RDF) types; edges also have one label, and zero or one type. PG node attributes become standalone nodes. Then, we leverage typed-strong summarization, just like for RDF graphs. This groups G_0 nodes according to their cliques, also taking care of nodes introduced by normalization instead of edges, etc. This is done with the same process as for RDF graphs (Section 5.2.4).

An alternative would be to summarize the property graph directly with the hierarchical summary technique [33] described in Section 3.2.7. However, this would only group PG nodes, not attribute values, which we view as data nodes (recall Section 4.1.5). Therefore, we would need a way to group those, e.g., by creating an equivalence class for each set of attribute value nodes for a given equivalence class of nodes and a given attribute name. As for RDF graphs, this would also require to cover PG edges, that have been transformed into data nodes while normalizing the data graph. Finally, following principle (■), the chosen partitioning method should provide a reasonable number of equivalence classes. This goal is not satisfied by [33], which tends to separate, in different classes, nodes with slightly different attributes. For these reasons, we prefer to use the typed-strong summarization.

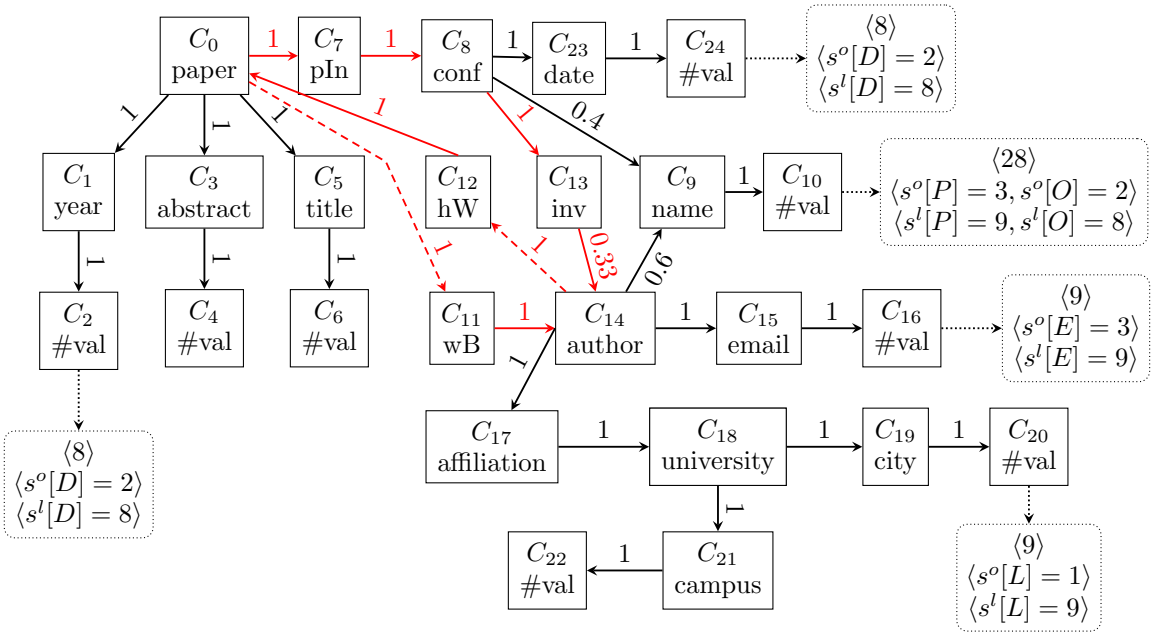


Figure 5.4: Collection graph corresponding to the data graph in Figure 4.5.

5.3 Collection graph and associated statistics

Based on the set of equivalence classes found as explained in Section 5.2, we build a directed **collection graph** $\mathcal{G} = (C, E_\mathcal{G})$ as follows:

- Each equivalence class (set of nodes) becomes a **collection node** C_i; we discuss collection nodes in more details in Section 5.3.1;
- For each data edge $n_i \to n_j$ in G such that $n_i \in C_i, n_j \in C_j$ are two graph nodes from two collections, the collection graph contains the **collection edge** $C_i \to C_j$. We detail collection edges and their properties in Section 5.3.2.

Each collection is an equivalence class, thus the collection graph is a quotient summary (Section 3.2.4).

Paths in the oriented collection graph are important for our analysis and understanding of this graph. We discuss the computation of various path properties in Section 5.3.3, and some simplifications (approximations) introduced by the collection graph in Section 5.3.4.

For illustration, we will rely on Figure 5.4, which shows the collection graph of the normalized graph presented in Figure 4.5.

5.3.1 Collection nodes

From each equivalence class computed as discussed in Section 5.2 we create exactly one **collection node** C_i (or collection, in short). A collection is thus a set of (G) nodes, and a $\mathcal{G}$ node itself; we also

say the collection **represents** each of its constituent graph nodes. Collection nodes are depicted as squares in Figure 5.4.

For each collection, we define:

Definition 5.3.1 (*Collection size*)

Given a collection node $C_i \in C$, its **size** $|C_i|$ is the number of data nodes it represents.

Definition 5.3.2 (*Collection label*)

The **label** of a collection node $C_i \in C$, denoted $L(C_i)$, is:

- The kind name, if all nodes in C_i have exactly one kind;
- The name of the most general kind, if C_i nodes have different kinds;
- Otherwise, when no kind is available, the label is:
  - `#val` if C_i represents $\mathscr{L}$-leaf data nodes;
  - The longest common prefix of the represented nodes' labels, if it is non-empty;
  - Otherwise, a concatenation of a very small set (a sample) of node labels.

For instance, in Figure 4.5, the label for nodes N_8, N_{19} and N_{24} is `paper`. If those nodes where typed, e.g., using RDF types such as `Article` or `Demonstration`, both being specifications of a more general type `ScientificPublication`, a good label for a collection containing them would be `ScientificPublication`. Alternatively, if the nodes are RDF URI resources, we attempt to find a significant common prefix of the URIs; this is helpful when the URLs are composed by adding numbers to a long string. This does not always succeed, e.g., on WikiData data where URIs are very short and include no meaningful word. In such cases, in best-effort mode, we sample a few node labels and show them to help users have an idea of what the nodes in the collection look like, e.g., `uri123,uri456,uri789`.

A collection node whose label ends with `#val` (recall Figure 5.4) is called a **leaf collection**:

Definition 5.3.3 (*Leaf collection*)

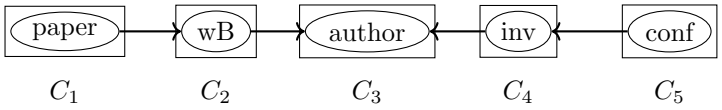
A collection node $C_i \in C$ is called a **leaf collection node** if it represents only data nodes whose labels belong to $\mathscr{L}$ (the set of literals).

Note that a childless node in the collection graph is not necessarily a leaf collection (as defined here). We reserve this name only for collections of $\mathscr{L}$-labeled nodes. By construction, such collections are guaranteed not to have outgoing edges in the collection graph.

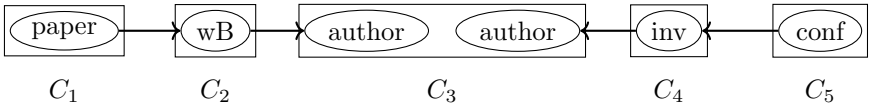
Because the collection graph is a (quotient) summary, like any summary, it introduces a certain loss of precision about how nodes and edges are connected. Below, we distinguish two particular configurations, when two collection edges have the same target in the collection graph:

Definition 5.3.4 (*Deeply shared collection*)

We say a collection C_z is **deeply shared** by collections $C_x, C_y \in C$, where C_x, C_y, C_z are three pairwise distinct collections, if there exists a data node $n_z \in C_z$, having both an incoming data edge from a node $n_x \in C_x$ and an incoming edge from a node $n_y \in C_y$.



(a) Deeply shared collections.



(b) Shallow shared collections.

Figure 5.5: Deeply shared vs shallow shared collections.

Definition 5.3.5 (*Shallow shared collection*)

We say a collection C_z is **shallow shared** by collections $C_x, C_y \in C$, such that C_x, C_y, C_z are pairwise distinct, if there exist two collection edges $C_x \to C_z$, $C_y \to C_z \in E_{\mathcal{G}}$, but C_z is not deeply shared by C_x and C_y.

Figure 5.5 illustrates both deeply and shallow shared collections. Each rectangle is a collection node, containing one or several data nodes. As shown in Figure 5.5a, C_3 is *deeply shared*: the node `author` is unique and shared by two distinct data nodes, specifically `wB` in C_2 and `inv` in C_4. Figure 5.5b shows that the collection of `author` nodes is *shallow shared*: `wB` and `conf` nodes are pointing to two distinct `author` nodes. The important difference here is that deeply shared collections feature common nodes (more than one incoming edge) at the data level, while shallow shared collections feature only common incoming edges (but they do not point to common data nodes). Looking at the collection graph alone does not allow to distinguish the difference.

We attach to each leaf collection node a **leaf entity profile** in order to represent which and how many Named Entities have been found in their string values. This will be useful both to classify abstraction entities in semantic classes (Section 6.8) and to enumerate entity paths (Section 7.3).

We call $\mathcal{T}$ **vector** an array of natural numbers indexed by the types in $\mathcal{T}$ (the set of Named Entity types considered in this work, recall Section 4.2). Let $|s|$ denote the length of a string s, s^o be the $\mathcal{T}$ vector of Named Entity occurrences in s, and s^l be the $\mathcal{T}$ vector of the total length of these entities. For instance, if s is "France and Germany are part of NATO", we have:

- $|s| = 34$, the string length;
- s^o[Location] $= 2$ (the countries France and Germany are extracted as Locations), s^o[Organization] $=$ 1 (NATO is identified as an Organization), and $s^o = 0$ for the other Named Entity types;
- s^l[Location] $= 13$, s^l[Organization] $= 4$, and $s^l = 0$ elsewhere.

The **string entity profile** of s is the triple $(|s|, s^o, s^l)$.

Next, for a collection C_i and a label l, we denote by $N^{C_i,l}$ the set of leaf children labels of an l-labeled child of a data node represented in C_i. For example, considering the collections drawn in Figure 5.4, $N^{C_{15},\texttt{name}} = \{$"Léa", "Eva", "Zoé"$\}$ (recall the normalized data graph in Figure 4.5). Finally, we define the **collection-label entity profile** as follows:

Definition 5.3.6 (*Collection-label entity profile*)

Given a collection $C_i \in C$ and a label l, the **collection-label entity profile** of C_i and l, denoted $\tau_{C_i,l}$, is the triple:

$$\tau_{C_i,l} = (\tau_{C_i,l}.s,\ \tau_{C_i,l}.o,\ \tau_{C_i,l}.l) = \left(\sum_{s \in N^{C_i,l}} |s|,\ \sum_{s \in N^{C_i,l}} s^o,\ \sum_{s \in N^{C_i,l}} s^l \right)$$

where the last two are component-wise $\mathcal{T}$ vector sums.

Figure 5.4 represents collection-label entity profiles in dotted rounded squares pointed by leaf collections. For instance, C_{10} collection-label entity profile shows that C_{10} leaf values are 28 characters long, they contain 3 Person NEs of 9 characters in total, and 2 Organization NEs of 8 characters in total. Note that $\tau_{C_i,l}.l$ differs from $\tau_{C_i,l}.s$ when extracted entities are shorter than the actual leaf value nodes. For instance, for the leaf value "VLDB conference", "VLDB" is extracted as an Organization, the Organization NE length is 4, whereas the string length is 15.

5.3.2 Collection edges

A collection edge **collection edge** $C_i \to C_j$ is created for any edge in G of the form $n_i \to n_j$, with $n_i \in C_i, n_j \in C_j$. We say that the (unique) collection edge $C_i \to C_j$ **represents** all the edges from a node $n_i \in C_i$ to a node $n_j \in C_j$.

We define:

Definition 5.3.7 (*Collection edge size*)

Given a collection edge $C_i \to C_j \in \mathcal{G}$, its **size** $|C_i \to C_j|$ is the number of data edges it represents.

Definition 5.3.8 (*Collection edge frequency*)

Given a collection edge $C_i \to C_j \in \mathcal{G}$, the **frequency** $f(C_i \to C_j)$ is defined as the ratio of data nodes in C_j having a parent in C_i, and the size of C_i. Formally:

$$f(C_i \to C_j) = \frac{|C_i \to C_j|}{|C_i|}$$

The collection edge frequency ranges in the interval $]0, +\infty]$.

For instance, in Figure 5.4, we have $f(C_{14} \to C_9) = 3/3 = 1.0$ because all authors have a name and $f(C_0 \to C_1) = 2/3 = 0.6$ because only two `paper` data nodes have a child node `year`. The frequency may be higher than 1 when several distinct nodes in C_j have the same parent in C_i. This is, e.g., the case of the edge $C_{14} \to C_{12}$, which has a frequency of $f(C_{14} \to C_{12}) = 4/3 = 1.3$, illustrating that N_{36} points to two `hw` data nodes.

Next, we introduce:

Definition 5.3.9 (*At-most-one collection edge*)

Given a collection edge $C_i \to C_j \in \mathcal{G}$, it is **at-most-one** if and only if each node in C_i is connected to at most one node in C_j.

For instance, in Figure 5.4, the collection edge $C_0 \to C_{11}$ is not at-most-one because, in Figure 4.5, N_{18} is connected to N_{21} and N_{23}, both represented by C_{11}. Similarly, the collection edge $C_{14} \to C_{12}$ is not at-most-one too. They are represented as dashed edges in the collection graph. This notion resembles 0-to-1 or 1-to-1 relationships in Entity-Relationship models [145].

We now consider how collection edges may be involved in cycles. Recall that a **cycle** is a non-empty *trail*, i.e., a sequence of edges which are all distinct, in which only the first and last nodes are equal. We define:

Definition 5.3.10 (*Acyclic collection edge*)

A collection edge $C_i \to C_j \in \mathcal{G}$ is **acyclic** if and only if it is not part of any cycle in $\mathcal{G}$.

For instance, collection edges that are involved in a cycle in $\mathcal{G}$ are in red in Figure 5.4. All the others are acyclic. Further, we define:

Definition 5.3.11 (*Data-acyclic collection edge*)

A collection edge $C_i \to C_j \in \mathcal{G}$ is **data-acyclic** if and only if all the G data edges it represents are not involved in any G cycle.

For instance, $C_0 \to C_{11}$ is not data-acyclic because the data edges it represents are also involved in a cycle in G, e.g., $N_{24} \to N_{28}$ and $N_{24} \to N_{30}$, also in red in Figure 4.5.

Next, we define the transfer factor of a collection edge. Intuitively, this measures how important (or frequent) C_i parents are among C_j nodes. Formally:

Definition 5.3.12 (*Edge transfer factor*)

The **edge transfer factor** of a collection edge $C_i \to C_j \in E_{\mathcal{G}}$, denoted $etf(C_i \to C_j)$, is the fraction of nodes from C_j having a parent in C_i. Formally:

$$etf(C_i \to C_j) = \frac{|\{n_j | \forall n_i, n_j \in N, n_i \to n_j, n_i \in C_i, n_j \in C_j\}|}{|C_j|}$$

The collection edge transfer factor ranges over $]0, 1]$.

For instance, in Figure 5.4, we have $etf(C_{12} \to C_0) = 2/2 = 1.0$ and $etf(C_{14} \to C_9) = 3/5 = 0.6$, while $etf(C_8 \to C_9) = 2/5 = 0.4$. Note how the collection C_9 *shares* its edge transfer factor among its parents C_8 and C_{14}. As shown in the examples, the edge transfer values may vary depending on the data nodes connectivity. Note that the edge transfer factor of a given collection edge is never 0, because, by definition, the corresponding collection edge exists due to at least one data edge connecting a node in C_i to a node in C_j. We distinguish several cases:

- **Between zero and one.** The edge transfer factor value lies between 0 and 1 (both excluded) when some C_j nodes are not pointed by any C_i node. For instance, Figure 4.5 shows that not all `name` nodes are pointed by `author` nodes, thus decreasing the edge transfer factor of the edge $C_{14} \to C_9$. A particular case is when C_j is a shallow shared collection. In Figure 5.4, `name` is a shallow shared collection between `author` and `conf`, thus splitting the edge transfer factor between them: $etf(C_{14} \to C_9) = 3/5$ and $etf(C_8 \to C_9) = 2/5$. The edge transfer factors of collection edges incoming C_j do not always sum to 1, e.g., when some C_j nodes are pointed by several data nodes of distinct collections connected to C_j. This corresponds to the case when C_j is a *deeply shared* collection.

- **One.** The edge transfer factor value equals 1 when at least each data node in C_j is pointed by one, or several, C_i nodes. A particular case is when each C_i node is connected to exactly one C_j node; this corresponds to an *at-most-one* collection edge.

5.3.3 Paths in the collection graph and their associated statistics

Abstractions and entity-focused path enumeration will need to rely on paths in a collection graph, and a set of associated statistics.

We define a **collection path** cp in a collection graph as a sequence of collection edges such that the source of each edge is the target of the previous edge in the path. We do not allow two edges in the same path to start in the same node (in other words, a path can be acyclic, or it can close a cycle, but if it closes a cycle, this is how the path ends):

Definition 5.3.13 (*Collection path*)

Given a collection graph $\mathcal{G}$, a **collection path** $cp \in \mathcal{G}$ is a sequence of collection edges such that $cp = C_i \rightarrow C_a, C_a \rightarrow C_b, ..., C_b \rightarrow C_j$.

The corresponding **collection path label** is the sequence of its nodes' labels, e.g., the path $C_{14} \rightarrow C_{15}, C_{15} \rightarrow C_{16}$ is labeled `author.email.#val`.

Algorithm 1 enumerates all the possible acyclic (collection) paths in the collection graph. It starts with paths of length $l = 1$, i.e., edges, and then iterates over the children of edge targets to build paths of length $l + 1$. In the end, it produces: $\mathcal{P}$, a 2D matrix containing all the possible acyclic paths connecting each pair of input graph nodes, and Θ, the set of input graph edges that are involved in a cycle (of size $l \geq 1$). Data structures are initialized in Lines 1-8, while iterative path enumeration takes place at Lines 9-25.

During the initialization phase, $\mathcal{P}$ is created as a 2D matrix indexed with G nodes on both dimensions. Each element in $\mathcal{P}$ is a set of paths (connecting the two indexed nodes), set by default to the empty set.

Next, the algorithm tries to extend each path in each $\mathcal{P}[i][j]$ with a new G edge of the form $j \rightarrow k$ (Line 19). If such an edge exists, we check whether k is equal to i to detect a potential cycle. If so, the current edge points back to the source of the path, leading to a cycle. Thus, we add all the edges in the extended path to Θ (Line 22). Otherwise, the newly extended path is added to $\mathcal{P}$ at the right indexes (Line 24).

Next, we define the **collection path transfer factor** for a given collection path cp, as follows:

Definition 5.3.14 (*Collection path transfer factor*)

Given a collection path cp, its **path transfer factor** $ptf(cp)$ is the product of all the edge transfer factors along that path. Formally:

$$ptf(cp) = \prod_{e \in cp} etf(e)$$

For instance, given the collection path $cp = C_8 \rightarrow C_{13}, C_{13} \rightarrow C_{14}, C_{14} \rightarrow C_9$ in Figure 5.4, its path transfer factor is $ptf(cp) = 1 * 0.33 * 0.6 = 0.198$.

Algorithm 1: Path enumeration and cycle detection algorithm

Input: $G = (N, E)$: a directed graph
Output: $\mathcal{P}$: complete set of paths in G, Θ: cycles in G

```
1  P ← [|N|][|N|]
2  Θ ← ∅
3  for i ∈ N do
4      for e of the form i → j do
5          if i == j then
6              Θ ← Θ ∪ {e}
7          else
8              P[i][j] ← P[i][j] ∪ {{e}}
9  stop ← false
10 l ← 1
11 while not stop do
12     l ← l + 1
13     stop ← true
14     for i, j ∈ N, i ≠ j do
15         paths ← P[i][j]
16         for p ∈ paths do
17             if |p| == (l − 1) then
18                 for e of the form j → k do
19                     p2 ← p ∪ {e}
20                     if i == k then
21                         for e2 ∈ p2 do
22                             Θ ← Θ ∪ e2
23                     else
24                         P[i][k] ← P[i][k] ∪ {p2}
25                         stop ← false
```

Based on the path transfer factor, we define the **total path transfer factor**, an important metric to elect entities to report in abstractions (Section 6.3). Intuitively, this represents how much of C_j weight can be transferred back to C_i by any collection path connecting C_i to C_j.

Definition 5.3.15 (*Total path transfer factor*)

Given two collections $C_i, C_j \in C$, the **total path transfer factor** $tptf(C_i, C_j)$ from C_i to C_j is the sum, over each path in $\mathcal{P}[i][j]$ (all collection paths connecting C_i to C_j that do not traverse an in-cycle edge), of the path transfer factor of that path. Formally:

$$tptf(C_i, C_j) = \sum_{p \in \mathcal{P}[i][j]} ptf(p)$$

For instance, in Figure 5.4, the path transfer factor between C_8 and C_{10} equals to $ptf(C_8, C_{10}) = 1 \times 0.4 = 0.4$.

If C_{13} had been involved in a cycle (which is not the case), then the total path transfer factor would have been $ptf(C_8, C_{10}) = 0.4 \times 1 + 1 \times 0.33 \times 0.6 \times 1 = 0.598$.

5.3.4 Discussion: simplifications made in the collection graph

We now discuss **approximations (simplifications)** introduced when building the collection graph. Grouping nodes by equivalence relations, such as those discussed earlier in Section 5.2, simplify the graph structure; this is beneficial because it allows to work on a much smaller collection graph than the original one. However, inevitable, such simplifications introduce some differences between the structures present in the original and the collection graph. We comment on two such cases below.

First, when, in the data graph, some nodes in collection `A` have children in the collection `B`, and some `C` nodes also have (different) `B` children, the resulting collection graph will contain a collection of `A` pointing to a collection of `B` and the collection of `C` pointing to the same collection of `B` even if they were disjoint in the data. In this case, the collection `B` is shallow shared by collections `A` and `C`.

Second, the collection graph may have cycles even when the data was acyclic. They are introduced when the label-based summarization is used on an XML document which features several elements with the same label at different places. For instance, the XML snippet in Figure 5.6 leads to the collection graph in Figure 5.7: note the cycle between `ul` and `li`.

5.4 From multiple datasets to a collection graph

In a heterogeneous dataset exploration scenario, it is very common to have several of these to interconnect on their *common values*. The graph view of any semi-structured data is the first step toward this goal (recall Chapter 4). The next one is to build a consolidated collection graph out of a data graph where many datasets co-exist, and possibly originate from different models. In this chapter, we have considered the collection graph of a single dataset, we now explain how this can be seamlessly extended to multiple datasets.

One could think that we can simply ingest all the data as explained in Chapter 4 and then build the collection graph. However, this is not possible because each data model has its own way to

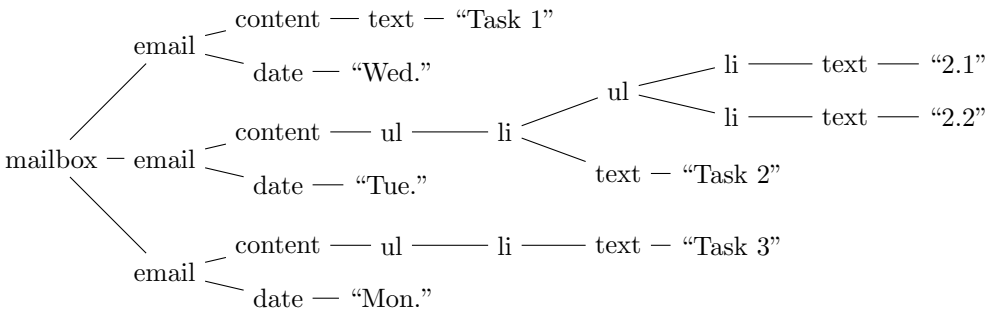


Figure 5.6: Data graph of an XML snippet about emails.

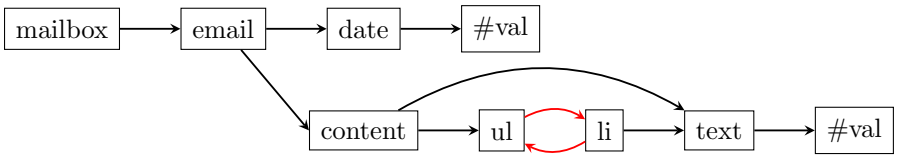


Figure 5.7: The collection graph corresponding to the normalized graph in Figure 5.6. Red edges are those involved in a cycle; those are also data-acyclic.

encode the *kind* (recall Section 5.1). Therefore, to obtain a single collection graph from a set of datasets, we proceed as follows:

1. First, we build a separate collection graph from each dataset, as described previously in the chapter.

2. Then, whenever two $\mathscr{L}$-leaf collections $C_i, C_j \in C$ from distinct datasets share values, we replace them by a new collection $C_{i,j}$, which contains the values of C_i and C_j, and inherits all the incoming and outgoing edges of C_i and C_j in the collection graph they came from. Their original collection graphs are thus connected.

3. The collection-label entity profile of each grandparent collection of a newly created collection $C_{i,j}$ is built.

For instance, the data graph in Figure 5.8 features RDF triples about NASA spacecrafts (on the right), and an XML document describing presidents who attended spacecraft launches (on the left). Named Entity nodes are highlighted, respectively in pink for Organization, yellow for Person and green for Location NEs; they are also connected to the value node they come from (dashed edges in Figure 5.8; recall Chapter 4). Note the three NEs found in the value node N_{17}, and the common NE "N. Armstrong" found both in the RDF and XML datasets values. Figure 5.9 shows the collection graph built out of the data graph in Figure 5.8 and reflects the *value unification*: (*i*) the `pilot` and `astronaut` leaf collections have been merged in a single collection C_9, representing N_{12} and N_{24}; and (*ii*) both entity profiles of (C_1, `pilot`) and (C_{11}, `astronaut`) lead to People NEs.

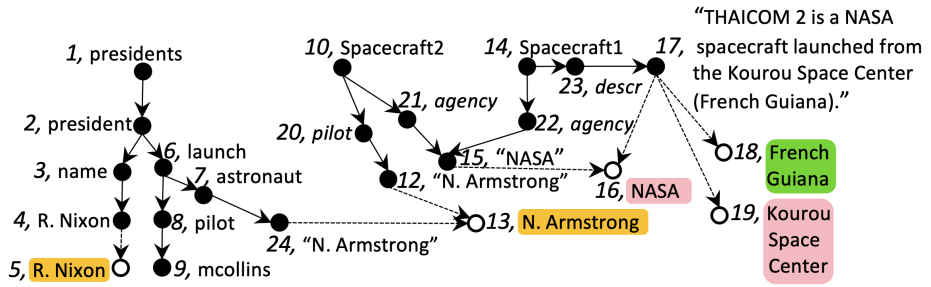


Figure 5.8: Sample normalized data graph for an RDF and an XML dataset about rocket launches.

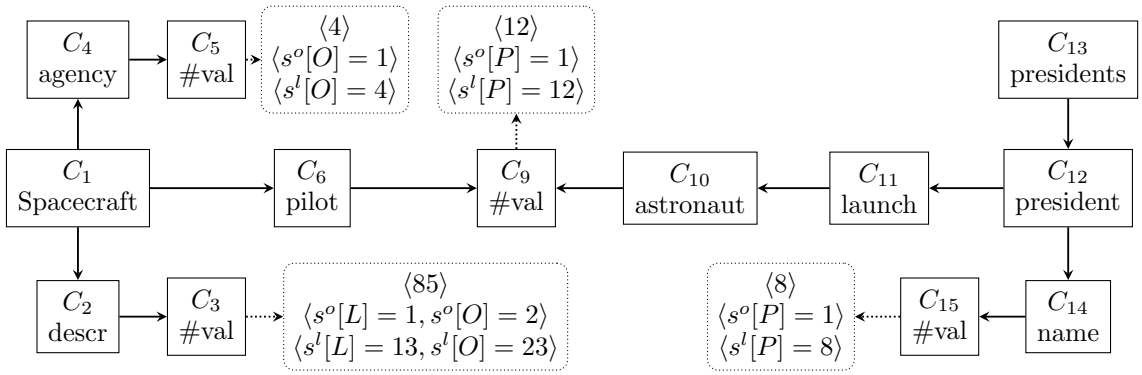


Figure 5.9: Multi-dataset collection graph corresponding to Figure 4.5.

5.5 Summary

This chapter has introduced a crucial technical ingredient of this thesis' result: the collection graph is a quotient summary of a data graph obtained from one or several individual datasets, each of which can be a relational, XML, JSON, RDF, or property graph dataset. Different equivalence relations are used for each of the data models; we have also shown how to weave summaries of individual datasets into a single integrated collection graph. Our collection graphs expose useful information about the presence of Named Entities extracted from the leaf nodes in the data. We also introduced a set of terminology referring to collection graph nodes and edges. All these features will be exploited by our next two chapters, outlining main technical contributions of the thesis.

6

Data abstraction

Chapter Outline

Chapter Abstract. We first explain how *entities*, *attributes* and *relationships* may appear in the collection graph (Section 6.1). Next, we provide algorithms to compute, from the collection graph, *main entities*, i.e., entities which deserve to be shown in the abstraction together with their attributes (Sections 6.2 and 6.7). Further, we present and detail several methods we devised for those algorithms (Sections 6.3, 6.4 and 6.5). Next, we show how to find main entities' relationships (Section 6.6). The last data abstraction step concerns the main entity classification into semantic classes (Section 6.8). Finally, we provide ABSTRA's experimental evaluation, the system we developed implementing all the above-mentioned steps (Section 6.9).

Data abstractions leverage the idea that any dataset comprises some *entities*, i.e., sets of records/data objects having attributes, which are often connected by *relationships*. In this chapter, our goal is to **retrieve, from the dataset, the conceptual model originally behind it**. However, to account for the recent adoption of complex, non-relational data formats, our abstractions differ from the classical Entity-Relationship schemas [145] in the sense that **our entities may have a deeply nested structure of attributes**. Finally, we seek to provide an abstraction *at the right level of details*, i.e., sufficiently compact to be readable by a human and sufficiently comprehensive to not lack important aspects of the data.

6.1 From the collection graph to entities

In a collection graph, some collections can be seen as sets of records, or "entities", while others contain "fields (or attributes)" of those entities, potentially nested several levels deep. For instance, in Figure 5.4, `paper`, `author` and/or `conf` can be viewed as entities; it appears natural that `year`, `abstract` and `title` would be attributes of the `paper` entity; similarly, `email` and `name` are attributes of the `author` entity.

In general, collection graphs may have hundreds of collections, and their structure may be quite complex (involve several cycles, etc.). Therefore, the selection of some entities as **main entities** may become complex. That is why we establish the two following requirements, which will guide our abstraction algorithm design:

(R1) We want the set of main entities to be rather small to not overwhelm the user with too much information.

(R2) Further, we want the main entities to *represent most of the data* since this is where the real interesting payload resides.

Toward determining main entities, their attributes and relationships in such graphs, our goals are to:

- Propose algorithms whose goals are to select a set of collections playing the role of the *most relevant* main entities. Further, the algorithm should be able to identify attributes which belong to each such main entity. Two algorithms, guided by requirements (R1) and (R2), are presented in Section 6.2. An alternative algorithm, devised at the end of the PhD, is proposed in Section 6.7 and lifts limitations of the first two algorithms. This global task may be divided into few smaller goals we enumerate here:

  (G1) Provide effective methods to assign scores to collections in order to decide which of them are the most relevant. Section 6.3 discusses a set of simple baselines as well as more elaborate methods, based on data weights and PageRank [136] scores.

  (G2) When a collection is identified as relevant, find which other collections may be seen as attributes of it. We call this task *entity boundary detection*, and provide algorithms for it in Section 6.4.

  (G3) Reflect that some main entities have been selected, and that, in turn, they (and the data they represent) are not available anymore for future steps. We call this task *graph update*, and we provide algorithms for it in Section 6.5.

- Identify all relationships between the main entities. We describe an algorithm for this task and provide an extension of it in Section 6.6.
- Assign a semantic category to each main entity to facilitate the dataset understanding, especially if the user is not familiar with the domain knowledge. The classification process is described in Section 6.8.

6.2 Main entity selection algorithm

In this section, we start by providing a definition for main entities, whose set is denoted $\mathcal{ME}$. Next, in Section 6.2.1, we provide a naive algorithm to select them and demonstrate its limitations. Further, we provide a greedy algorithm in Section 6.2.2.

A **main entity** is composed of a **root** (the collection that is identified as interesting and which contains records) and a **boundary** which may be viewed as comprising *entity's (deeply nested) properties*. Formally:

Definition 6.2.1 (*Main entity*)

We call **main entity** $\mathcal{ME}_i$ a collection $C_i \in C$, together with a sub-graph of the collection graph, rooted in C_i, denoted $\mathcal{B}_i$, and called *boundary* of the main entity. Each collection graph node that is part of $\mathcal{B}_i$ is reachable from C_i by one or several paths, whose nodes and edges are also in $\mathcal{B}_i$.

Conceptually, the boundary includes all the data nodes that are part of the main entity (or, equivalently, "belong to" the main entity root). However, our boundaries are defined on the collection graph $\mathcal{G}$, not the normalized data graph G. For instance, Figure 6.1 depicts three possible main entities in $\mathcal{G}$: `author`, `paper` and `conf`. Each main entity root is colored in yellow. In this abstraction, the `conf` main entity has two attributes, a `name` and a `date`. The `paper` one has three attributes: `year`, `abstract` and `title`. Finally, the main entity containing `author` records is the deepest one: it contains six attributes, nested in three levels. The highlighted areas in blue, green and red, represent the *boundary* of each main entity. We will discuss how to compute boundaries, in Section 6.4.

Next, leveraging classical E-R design [145], we identify a set of **main entity eligibility criteria** a collection must satisfy to be considered as a potential main entity:

(C1) It should represent more than one data node, discouraging one-node collections as degenerate "entity sets". Also, the node in such a singleton collection is typically a container to many nodes from another, potentially interesting, collection.

(C2) It must have at least two child collections and/or at least one child collection having a leaf child. This adapts to our setting the intuition that "entities have attributes":

- childless collections do not qualify because they have no internal structure;
- collections with a single child that is a leaf do not qualify for the same reason;
- collections with a single, non-leaf child do not qualify; they act more as "containers" for their children.

(C3) It should not already be part of a boundary of another main entity.

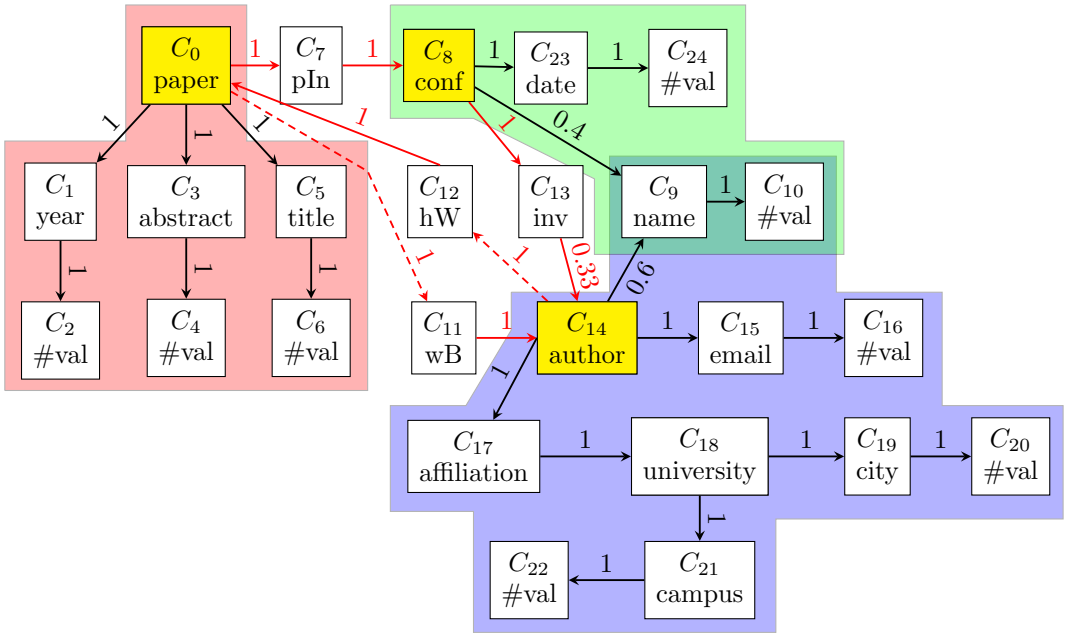


Figure 6.1: The collection graph with 3 possible main entities and their boundaries.

We illustrate the above criteria on few XHTML search results, grouped in pages, in Figure 6.2. First, the (only) `top` element is the parent, or *container*, of all (more interesting) `page` nodes. Therefore, it is not eligible according to (C1). Next, following (C2), all leaf or parents of leaf collections, such as those containing `date` nodes or `text` value nodes, do not qualify. Similarly, the `page` collection has a single child, namely `result`, with the actual data; the `result` collection is eligible, but the `page` one is not as per (C2).

In a collection graph, one may identify more or less entities, depending on its structural complexity. We consider a **target entity number** E_{max}, which the users can configure, and seek to solve the problem (informally) defined as follows: identify, from the collection graph, the E_{max} most interesting entities (if there are at least E_{max}), otherwise, return all that can be found. As an

```
<top>
  <page>
    <result><date>01/01/2021</date><text>My first result</text></result>
    <result><date>02/01/2021</date><text>My second result</text></result>
    ...
  </page>
  <page> ... </page>
  ...
</top>
```

Figure 6.2: An XHTML search results, grouped in pages.

alternative, instead of specifying E_{max}, users could specify a *minimum coverage*, where we define coverage as: the fraction of nodes from the normalized graph, that are comprised in one of the returned main entities. We denote the user-specified coverage score threshold by cov_{min}.

6.2.1 Naive algorithm

A naive way to produce the set of main entities is to assign to each collection a *score* (or *weight*) cw, and to select the E_{max} collections having the highest scores. Algorithm 2 formalizes this idea; we will discuss concrete score alternatives in Section 6.3. First, we initialize cws, an array indexed on the collection ids and storing the score of each collection (Line 1). Further, for each collection $C_i \in C$, we compute its *score*. (Lines 2-3). Next, we iteratively select at most E_{max} entities or until the coverage of the selected main entities reaches cov_{min} (Lines 4-10). Those two stopping criteria help follow requirement (R1), stating that the size of $\mathcal{ME}$ should be rather small (recall Section 6.1). At each step, we select C^*, the next *best* collection (Line 5), using Algorithm 3. It takes as input cws, the array of collection scores, and returns the identifier of the next *best* collection. First, we fill *cands* with the set of eligible collections (recall eligibility criteria above) maximizing the score (Lines 1-4). If there is no candidate, we return -1 to signal it (Line 5). If there is only one candidate, we return it (Line 7). Otherwise, several candidate collections have the highest score, therefore a tie occurs. We use the following heuristics to determine which collection to select among the best candidates:

(H1) The collection with the highest number of children wins the tie (Lines 10-13).

(H2) If (H1) is not sufficiently restrictive, the largest collection in terms of size wins (Lines 15-18).

(H3) If (H1) nor (H2) could help, we select the collection with the lowest id (Line 20)[1].

Algorithm 2: Naive main entity selection algorithm

Input: $\mathcal{G}$: a collection graph
Output: $\mathcal{ME}$: the set of main entities found in $\mathcal{G}$

```
1  cws ← [ ]
2  for C_i ∈ C do
3      cws[C_i] ← score(C_i)
4  while |ME| ≤ E_max or cov < cov_min do
5      C* ← nextCollection(cws)
6      if C* = −1 then
7          stop
8      else
9          b ← boundary(C*)
10         ME ← {(C*, b)} ∪ ME
```

If Algorithm 3 could find a candidate for the next main entity, we create it as a new main entity in $\mathcal{ME}$, i.e., a pair composed of $C^*$for the root and the boundary computed by any method described in Section 6.4.1, 6.4.2 or 6.4.3.

[1]We could take any other heuristic, e.g., the collection with the largest id; the important thing is to make a deterministic choice and do not select a collection randomly.

Algorithm 3: Next collection selection method

Input: *cws*: an array of scores
Output: *id*: a collection id

```
1  cands ← {}
2  for cand ∈ argmax_all(cws) do
3      if isEligible(cand) then
4          cands ← cands ∪ {cand}
5  if |cands| = 0 then
6      return −1
7  else if |cands| = 1 then
8      return cands[0]
9  else
10     children ← count_children(cands)
11     winnerNbChildren ← argmax_all(children)
12     if |winnerNbChildren| = 1 then
13         return winnerNbChildren[0]
14     else
15         nodes ← count_nodes(cands)
16         winnerNbNodes ← argmax_all(nodes)
17         if |winnerNbNodes| = 1 then
18             return winnerNbNodes[0]
19         else
20             return cands[min(cands.indexes)]
```

6.2.2 Greedy algorithm

The naive approach to select main entities (Algorithm 2 in Section 6.2.1) is rather limited because it does not reflect that some collections have already been reported. Indeed, collection scores are computed only once (at the beginning), thus the naive algorithm always decides what is the next main entity based on the initial set of scores, which becomes gradually obsolete as long as main entities are selected.

Therefore, we propose a *greedy* approach, leveraging the intuition that making locally optimal choices at each step of the algorithm will produce more sound results. The idea of this greedy approach is implemented through Algorithm 4. The major difference with the naive approach (Algorithm 2) lies in the graph update and scores re-computation after each main entity selection. The intuition behind the graph update task (Line 11) is to identify the next main entity as if *"all the data nodes and edges that are part of the previously selected entities where not in the data"*. Indeed, previously selected main entities are already part of $\mathcal{ME}$, thus they are not useful anymore in the decision of the next best collection to select. Making graph updates after each main entity selection involves a score re-computation because some data is now *excluded* from the data graph, thus the collection graph will change (Line 13). Note that the rest of the algorithm is the same as in Algorithm 2.

Algorithm 4: Greedy main entity selection algorithm

Input: $\mathcal{G}$: a collection graph
Output: $\mathcal{ME}$: the set of main entities found in $\mathcal{G}$

```
1  cws ← [ ]
2  for C_i ∈ C do
3      cws[C_i] ← score(C_i)
4  while |ME| < E_max or cov < cov_min do
5      C* ← nextCollection(cws)
6      if C* = −1 then
7          stop
8      else
9          b ← boundary(C*)
10         ME ← {(C*, b)} ∪ ME
11         graphUpdate()
12         for C_i ∈ C do
13             cws[C_i] ← score(C_i)
```

6.3 Scores

In Section 6.2, we presented two algorithms (Algorithms 2 and 4) to produce $\mathcal{ME}$, the set of main entities. Those algorithms both rely on scores assigned to collections, which can be computed in several ways. We first define simple scores in Section 6.3.1, and we discuss their limitations. Next, we define more elaborate scores, based on a notion of data weight in Section 6.3.2, and the well-known PageRank algorithm [136] in Section 6.3.3.

6.3.1 Simple collection scores

We start with considering some very simple scoring techniques.

Roots

Following the intuition that "children collections belong to their parents", e.g., `title` belongs to `paper`, the **root** method assigns a score of 1 for root collections, 0 to others. Formally, we have:

Definition 6.3.1 (*Root score*)

Given a collection $C_i \in C$, its **root score** is: $\begin{cases} 1 & in\_degree(C_i) = 0 \\ 0 & \text{otherwise} \end{cases}$

Such a score would suggest the root(s) of the collection graph as main entities. However, this may be inapplicable: the collection graph may have no root if it is cyclic, as in Figure 5.4.

Node count

We could select collections with the highest **number of nodes** (recall collection size in Section 5.3.1), based on the intuition that they cover a large part of a dataset. This may also be inappropriate in some cases, e.g., in Figure 5.4, the `name` collection is the largest, because it contains both people names and conference names, but `name` says very little of what the dataset is about. Moreover, in case of regular data, when each record tends to have one child of a given kind, the collection containing the records will have the same score as a collection containing children of these records. For instance, in Figure 4.5, all papers have a title; thus, there are as many `title` nodes and `title` value nodes as `paper` nodes. This score does not give a sufficiently strong signal to help selecting main entity roots.

Child count

Another intuition is that important entities are likely to have many attributes. Thus, we can select collections having the largest number of children in the collection graph. We define this score as follows:

Definition 6.3.2 (*Child count score*)

Given a collection $C_i \in C$, its **number of children score** is $|\{C_j | C_i \rightarrow C_j \forall C_i, C_j \in C\}|$.

For instance, C_{14}, the `author` collection, has four children: `name`, `email`, `affiliation`, and `hW`, thus obtains a score of 4. This method, however, does not account for deeply nested structure, e.g., the affiliation has several nested children.

Descendant count

In order to account for potentially deeply nested attributes, we can select collections having the largest number of descendants. We call this method $desc_k$ and define it as follows:

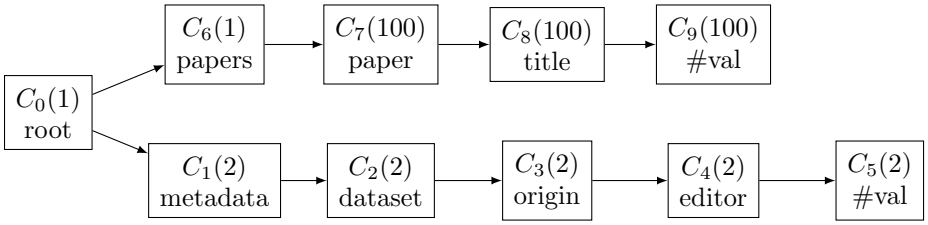


Figure 6.3: A collection graph showing $desc_k$ limitations.

Definition 6.3.3 (*Descendant count score*)

Given a collection $C_i \in C$ and an integer $k > 0$, its **number of descendants score at depth** k is:

$$desc_k(C_i, k) = |descendants(i, k)|$$

considering that the descendants of a node at a depth k is defined as follows:

Definition 6.3.4 (*Limited-depth descendant count*)

Given any graph, the **descendants of a node** i **at depth** k is the set of nodes j such that there exists a path in the graph from i to j without any in-cycle edge and containing at most k edges, i.e.,

$$descendants(i, k) = \forall j \in [0, |N|[, \forall p \in P[i][j] s.t. |p| \leq k, \bigcup_{e \in p} \{C_i, C_j\}$$

In practice, we have experimented with $k \in \{1, 2, 3\}$, leading to the scores $desc_1$, $desc_2$, $desc_3$. However, a score based on the number of descendants may rapidly become unfair depending on the dataset structure. Indeed, it will favor collections with deep structures, even if they do not carry much of the data. For instance, Figure 6.3 shows a collection graph obtained from an XML document describing scientific publications. For each collection, we show its id, name and size (the number in parenthesis). $desc_3$ would select the collection C_1 of `metadata` nodes. Observe that C_0 is not eligible because it represents only one node and C_6 is also not eligible because it acts as a container. In this example, $desc_3$= 4 for `metadata`, while it values only 2 for `paper`. Nevertheless, this does not seem to be a wise choice, especially considering the information brought by the 100 papers. It also contradicts our requirement (R2) to select main entities representing most of the dataset (recall Section 6.1).

Leaf descendant score

Instead of counting all descendants, we can count only leaf descendants. This variant, denoted $leaf_k$, for some integer k, relies on the assumption that data content in leaf (value) nodes (paper titles, paper author names, etc.) is more important than the structural nodes which only serve to "organize" the values. $leaf_k$ assigns to a collection a score equal to the number of leaf collections reachable at distance at most k. Similarly, $k \in \{1, 2, 3\}$ leads to the score functions $leaf_1$, $leaf_2$, $leaf_3$, respectively.

Definition 6.3.5 (*Leaf descendants score*)

Given a collection $C_i \in C$ and an integer $k > 0$, its **leaf descendants score at depth** k is:

$$leaf_k(C_i, k) = |\{n | \forall n \in descendants(C_i, k) \wedge isLeaf(n)\}|$$

From now on, we only consider the descendant count score $desc_k$, and the score based on the leaf descendant count, $leaf_k$. This is because other scores do not reflect at all the data depth and collection graph complexity, thus contradicting our requirement (R2).

6.3.2 Scores by DAG propagation

In Section 6.3.1, we have defined methods mostly accounting the structure of the data (*how many children?, how many descendants?, etc.*). However, data and collection graphs may have various shapes and may not be always well-balanced. Therefore, and to meet requirement (R2), specifying that the final set $\mathcal{ME}$ should reflect *where the data is*, we define more elaborate scores, based on the notion of data weight, introduced below.

We start by defining the own data weight of a node as follows:

Definition 6.3.6 (*Node own data weight*)

Given a node $N_i \in G$, its **own data weight** $ow(N_i)$ is the number of edges incoming n:

$$ow(N_i) = in\_degree(N_i)$$

In tree data, ow is equal to 1. Otherwise, ow may be greater than 1. This is the case for RDF literals that are the value of several triples, XML nodes being the identifier of some references (recall ID-IDREF connections in Section 4.1.2), and JSON maps in collections that have been merged due to their common keys (recall Section 4.1.3).

We can now define a collection's own data weight as follows:

Definition 6.3.7 (*Collection own data weight*)

Given a collection $C_i \in C$, its **collection own weight** $ow(C_i)$ is the sum of the own data weights of nodes it represents:

$$ow(C_i) = \sum_{N_j \in C_i} ow(N_j)$$

We have devised an algorithm, called w_{DAG}(Algorithm 5), in which leaf collections *transmit their own data weight back to all their ancestors*, along ancestor-descendant paths that do not overlap with cycles in the collection graph. Before applying w_{DAG}, we proceed to some initialization. First, we compute the ow of each $\mathscr{L}$-leaf data node $N_i \in G$. Leaf structural nodes, e.g., RDF resource nodes without outgoing edges, XML elements without children, or JSON empty maps or arrays, get an ow of 0, following requirement (R2). Then, each $\mathscr{L}$-leaf collection node gets its ow computed

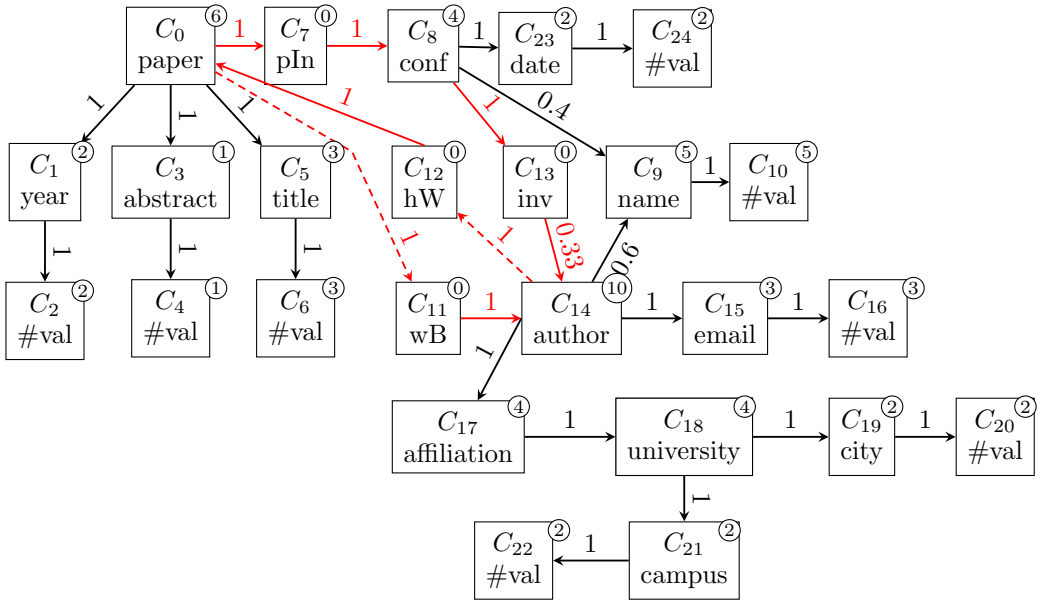


Figure 6.4: The w_{DAG} propagation on the collection graph.

as the sum of leaf nodes *ow* in that collection. Next, the **collection weight of a collection** C_i, $cw(C_i)$, is initialized to $ow(C_i)$ for $\mathscr{L}$-leaf collection nodes, 0 for others. Algorithm w_{DAG} leverages the total path transfer factor (recall Section 5.3.3) between a leaf collection and one if its ancestors (Line 3). A collection i gets its $cw(i)$ increased only by acyclic paths connecting it to one of its descendant collections k. It is worth noting that the highest the $tptf$, the more $\mathscr{L}$-leaf collection weights are reflected in their ancestors.

Algorithm 5: w_{DAG}

Input: $\mathcal{G}$: a collection graph
Output: *cw*: a set of scores for each node in $\mathcal{G}$

```
1 foreach ℒ-leaf collection C_k ∈ 𝒢 do
2     foreach collection C_i ∈ 𝒢 do
3         cw[C_i] ← cw[C_i] + ow[C_k] * tptf[C_i][C_k]
```

Figure 6.4 shows the w_{DAG} propagation on our collection graph in Figure 5.4. For each collection node, we indicate, in the attached circle, its collection weight *cw*. Observe how collections having only outgoing edges in cycles, i.e., $C_7, C_{11}, C_{12}, C_{13}$, have a *cw* of 0. This is because they cannot get any data weight from any leaf collection.

The main limitation of w_{DAG} is its failure to propagate scores through cyclic edges. For instance, in the collection graph of Figure 5.7, the w_{DAG} collection weight of the collection labeled `content` score does not reflect all the `text` values *ow* because some paths connecting these two collections contain collection edges involved in cycles. In particular, only the `text` values that have a grandparent

labeled `content` will be reflected. Note that cycles are very frequent in graphs, e.g., in RDF and property graphs, thus they may appear in the respective collection graphs; also, the collection graph may be cyclic even if the data is tree-structured (recall Section 5.3.4).

6.3.3 Scores using PageRank

To escape the main limitations of simple and the w_{DAG} scores, we have devised two scoring methods based on **PageRank** [136], a well-known algorithm used to compute weights (or scores) in a graph based on propagation along edges, regardless of the structure of the graph. PageRank has been initially introduced by Google's search engine in order to rank Web pages, and thus Web search results.

The main intuition of PageRank is the following: given a directed graph, a node is itself important if it is pointed by a large number of important nodes (note the recursive definition). In the context of Web searches, the PageRank score of a Web page n is the probability that a random user will arrive on n after a large number of random traversal steps on the graph, following Web links as directed graph edges. Indeed, the more important a page is, the higher the probability a user reaches it is. Importantly, PageRank scores reflect *only* the graph topology and are **independent** of any context, such as a Web query or the graph node weights.

Algorithm 6 recalls the PageRank algorithm from the literature [3]. Given a directed graph with a set of initial node weights and a set of edge weights, it computes the PageRank score of each node in that graph. PageRank propagates initial node weights in an iterative manner, meaning that node weights are propagated to their direct neighbors at each iteration. PageRank stops its propagation when node weights have stabilized or after a certain number of iterations. Formally, the algorithm takes two inputs: P, a square matrix of edge weights, and U, a square matrix of node weights. Both P and U are computed based on an input directed graph and are both indexed on that graph nodes (see below for P and U initialization). The algorithm yields v, a vector of PageRank scores, one for each node in the graph. The algorithm starts with main structures initialization (Lines 1-4):

- v is a vector indexed on node ids and stores the current PageRank score of each node;
- $oldV$ is a copy of v and stores the PageRank scores of the previous iteration;
- M is a square matrix, of same dimensions as P, and is used to compute matrix multiplications at a given iteration;
- d is a probability between 0 and 100%, typically set to 15%, modeling the probability to randomly jump to another node in the graph;
- $iter$ is an integer counting PageRank iterations;
- k is the maximum number of iterations allowed before PageRank stops (typically set to 100);
- $maxChange$ is the maximum difference observed for each $v[i]$ and $oldV[i]$ for any $0 \leq i < |v|$;
- σ is the change threshold, typically set to 10^{-6}, under which convergence is declared.

Next, we compute M, the square matrix modeling the different choices the algorithm can make at each iteration (Line 5). It can follow graph edges outgoing the current node, with a probability of $1 - d$; this corresponds to the left arm of the sum in Line 5 and allows to naturally handle graph cycles because node weights are propagated on directed neighbors only. Otherwise, with a

probability of d, it can jump to another random node in the graph, such that each node in the graph has a uniform probability to be chosen as the destination of the jump. This corresponds to the right arm of the sum and helps PageRank to visit all parts of the graph, even the disconnected ones. Rootless and multiply-rooted graphs are also of no problem for PageRank as weights are propagated all over the graph and in all directions. The iteration process can now start for at most k iterations or until convergence (Line 6), defined as the point where a new iteration brings only a very small change to the vector of node scores. A standard value for PageRank convergence is $\sigma = 10^{-6}$, that is, scores should not change by more than σ of their value. Each iteration repeats four steps. First, the matrix v of current PageRank scores is stored in $oldV$. Then, v is updated by multiplying it by M, the transition matrix saying whether to follow graph edges or jump to random nodes. Then, the change between each element in v and $oldV$ is computed; the maximal value is kept in $maxChange$. Finally, the number of iterations is increased by 1.

Algorithm 6: PageRank

Input: U: a matrix of node weights, P: a matrix of edge weights
Output: v: a vector of PageRank scores

1 $v, oldV \leftarrow [\,]$
2 $M \leftarrow [\,][\,]$
3 $d \leftarrow 0.15$; $iter \leftarrow 1$; $k \leftarrow 100$; $maxChange \leftarrow 1$
4 $\sigma \leftarrow 0.000001$
5 $M \leftarrow (1-d) \times P^T + d \times U$
6 **while** $iter \leq k \wedge maxChange \geq \sigma$ **do**
7 $\quad oldV \leftarrow v$
8 $\quad v \leftarrow M \times v$
9 $\quad maxChange \leftarrow maxValue(v - oldV)$
10 $\quad iter \leftarrow iter + 1$

PageRank-based scoring

Our first PageRank-based scoring, denoted w_{PR}, leverages the PageRank algorithm. Algorithm 7 details how w_{PR} works; it can be applied on any graph g, including G and $\mathcal{G}$.

First, the matrices U and P of node and edge weights are computed following the literature [3]. For **node weights** (Lines 2-4), U is filled with the value $\frac{1}{|N|}$ (for all cells). This models the fact that users may start their walk on any node in the graph with a uniform probability. The initial value used to set any $U[i][j]$ should be wisely defined in order to let the algorithm converge as rapidly as possible. Indeed, one can set this value to any positive value, e.g., 0.1, 1, 1000, the further the initial value is from the optimal value, the longer the convergence is. Therefore, the literature recommends initializing node weights to $\frac{1}{|N|}$. For **edge weights** (Lines 5-10), it goes as follows. For two nodes $N_i, N_j \in g$, $P[i][j] = \frac{1}{|out\_degree(N_i)|}$ if there exists an edge from N_i to N_j, 0 otherwise. This initialization ensures two properties: (i) the matrix is stochastic, i.e., all rows sum to 1, and (ii) all N_i's neighbors have a uniform probability to be chosen when the random user is on N_i. The PageRank initialization for edge weights could stop here, but a problem arises with nodes having no outgoing edges (typically called **sink nodes**): such nodes cannot propagate any weight. The PageRank algorithm treats each sink N_i as follows: $|N|$ artificial edges are created,

connecting N_i to each $N_j \in g$ and their edge weight is $P[i][j] = \frac{1}{|N|}$ (Lines 11-13). This models a uniform probability to move to any other node in the graph for all sink nodes.

Algorithm 7: w_{PR}

Input: $g = (N, E)$: a graph
Output: *scores*: a set of scores for each node in g

```
1  P, U ← [ ][ ]
2  for i ∈ {0, 1, ..., |N|} do
3      for j ∈ {0, 1, ..., |N|} do
4          U[i][j] ← 1/|N|
5  for i ∈ {0, 1, ..., |N|} do
6      for j ∈ {0, 1, ..., |N|} do
7          if N_i → N_j ∈ g then
8              P[i][j] ← 1/out_degree(N_i)
9          else
10             P[i][j] ← 0
11     if out_degree(N[i]) = 0 then
12         for j ∈ {0, 1, ..., |N|} do
13             P[i][j] ← 1/|N|
14 scores ← PageRank(U, P)
```

We apply Algorithm 7 on the collection graph with **reverse edges**, denoted $\mathcal{G}_R$, in order to let leaf collection nodes propagate their importance to non-leaf collection nodes. Finally, the set of rankings that PageRank computed is exactly the set of collections scores (Line 14). Figure 6.5 shows the w_{PR} propagation applied on the reverse collection graph (note the inverse collection graph edges); numbers on edges are PageRank edge weights. The circle attached to each collection node shows the w_{PR} score, shorten after 3 decimals. Note how two nodes at the same "position" in the graph obtain the same score. For instance, leaf collection nodes, e.g., C_2, C_4 and C_{22}, all obtain a score of 0.006. Similarly, parent collections of leaf collections all obtain a score of 0.011. Note that the sum of collection weights equals 1. Finally, observe that, starting from a collection node, there is equal chance to walk on each outgoing edge. For instance, from C_9, there is equal chances to go to C_8 and C_{14}.

Data-weighted PageRank-based scoring

The above method presents a score based solely on the graph topology (recall how nodes at the same position in the graph have same PageRank score and how there are equal chances to go on all neighbors of a node). However, the data weight, as described in Section 6.3.2, plays an important role in the task of assigning good scores to graph nodes, typically collection nodes. Also, not using it contradicts our requirement (R2) of selecting entities which cover a large part of the dataset (recall Section 6.1). Therefore, we devise a new method, namely w_{dw-PR} (*data-weighted PageRank*), which addresses this problem and which applies to the reverse collection graph $\mathcal{G}_R$ (not to any graph, as for w_{PR}). For that method, one could think of initializing the PageRank node weights with ow,

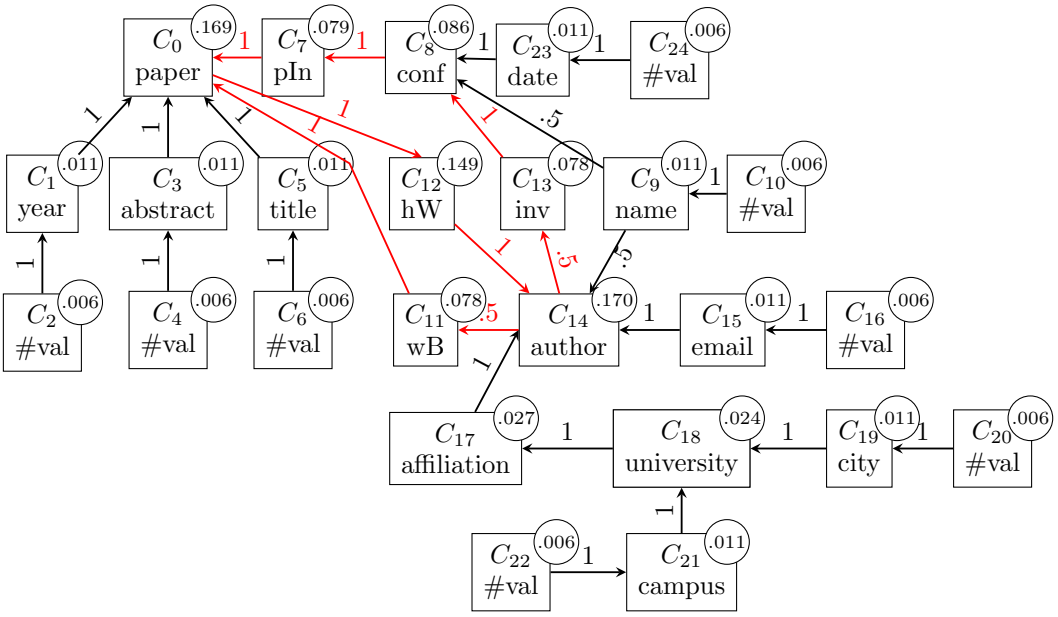


Figure 6.5: The w_{PR} propagation on $\mathcal{G}_R$, the reverse collection graph.

as defined in Section 6.3.2, and then to let PageRank move them until convergence. However, this would not work because PageRank final node scores are independent of the initial ones, thus data weights would not be reflected in the final collection scores. Instead, we reflect them in edge weights; this is why w_{dw-PR} resembles Algorithm 7 in the way it initializes node weights, but not edge weights. We present w_{dw-PR} in Algorithm 8. It goes as follows:

1. First, each collection node in $\mathcal{G}_R$ obtains a node weight of $\frac{1}{|C|}$ (Lines 2-4), as in the literature.

2. Next, in Line 8, for each collection $C_i \in \mathcal{G}_R$ *with* some outgoing edges, we set the weight of each outgoing edge $C_i \to C_j$ as: the number of data edges represented by the corresponding collection edge in $\mathcal{G}$, i.e., $C_j \to C_i$, divided by the number of data edges outgoing C_i in $\mathcal{G}_R$ (corresponding to C_j in $\mathcal{G}$). Observe that these weights sum to 1 for each $C_i \in \mathcal{G}_R$, following the PageRank convention.

3. Otherwise, for each collection $C_i \in \mathcal{G}_R$ *without* outgoing edges, i.e., sink collection nodes, w_{dw-PR} creates an edge going from C_i to every other node $C_j \in \mathcal{G}_R$ (Line 11). We assign to such an "artificial" edge, a weight obtained by dividing the number of nodes in C_i by the total number of nodes in the graph (artificial edges are also inverted compared to the collection graph and data edges). Such weights also sum up to 1 for each C_i, following the PageRank convention.

Observe that we compute G_R edge weights from edge (not node) counts. To see why, consider an address shared by two companies (thus, two incoming edges). Intuitively, each company has this address, not just half of it. More generally, shared nodes should weigh as many times as they are shared; this is reflected by counting incoming edges.

Algorithm 8: w_{dw-PR}

Input: $\mathcal{G}_R$: a reverse collection graph
Output: cw: a set of scores for each collection node in $\mathcal{G}_R$

$P, U \leftarrow [\,][\,]$
for $i \in \{0, 1, ..., |C|\}$ **do**
  for $j \in \{0, 1, ..., |C|\}$ **do**
    $U[i][j] \leftarrow \frac{1}{|C|}$
for $i \in \{0, 1, ..., |C|\}$ **do**
  if $out\_degree(C_i) > 0$ **then**
    for $j \in \{0, 1, ..., |C|\}$ **do**
      $P[i][j] \leftarrow \frac{|C_j \to C_i \in \mathcal{G}|}{|C_j \to C_k \in \mathcal{G}|}$
  else
    for $j \in \{0, 1, ..., |C|\}$ **do**
      $P[i][j] \leftarrow \frac{|C_i|}{|C|}$
$cw \leftarrow PageRank(U, P)$

Figure 6.6 shows the reverse collection graph and the corresponding PageRank scores in the circles attached to collection nodes. Observe how edge weights are not all equal for a given collection node, but are still summing to 1. For instance, starting from C_9, PageRank has more chances to go on C_{14}, than on C_8. This is how the data-weight is reflected by w_{dw-PR}: more authors have names than conferences. Same applies for C_{14}, C_{11} and C_{13}. Final node weights reflect this: the `author` collection has a larger weight than it has with w_{PR} (see Figure 6.5); similarly, the `conf` collection has a lower weight than with w_{PR}. When the collection graph contains hundreds or thousands of collections and/or the dataset is unbalanced, w_{dw-PR} is able to reflect better where the data is than w_{PR}, as shown in the experimental evaluation in Section 6.9.

6.4 Boundary methods

After assigning scores to collections, the next step in our main entity selection algorithms (Algorithms 2, 4 and 11) is to compute their **boundaries**. Recall that boundaries correspond to the sub-graph of collections that "belong to" the main entity. The boundary computation depends on the method used to assign scores: how collections obtain their scores naturally defines a boundary. Therefore, we present boundaries for simple scores in Section 6.4.1, for data weights in Section 6.4.2 and for PageRank-based scores in Section 6.4.3.

6.4.1 Boundaries for simple scores

Given a main entity root C^* elected using $desc_k$ or $leaf_k$ methods, its natural boundary represents all the collection nodes, respectively leaf collection nodes, in the descendants of C^* far from at most k edges from C^*. We denote such a boundary $bound_{desc}$, respectively $bound_{leaf}$.

For instance, in the collection graph in Figure 5.4, the $bound_{desc}$ of C_{14}, when the score is $desc_2$,

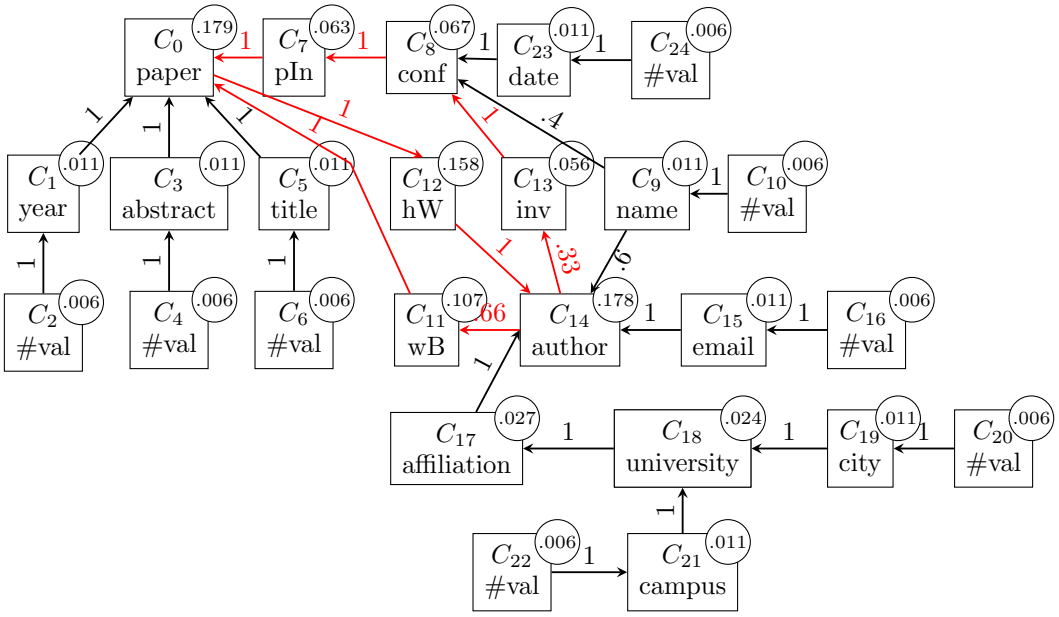


Figure 6.6: The w_{dw-PR} propagation on $\mathcal{G}_R$, the reverse collection graph.

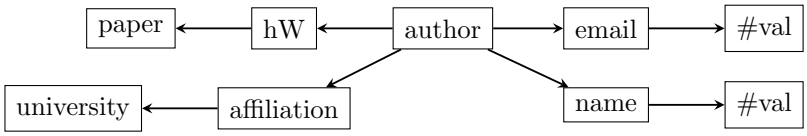


Figure 6.7: The $bound_{desc}$ boundary obtained for the main entity `author`, based on $desc_2$ scores.

is the (sub-)graph shown in Figure 6.7. One can observe how such boundaries take exactly the collections that contributed to the main entity. However, they do not take into account the depth of the data, e.g., the paper main entity is published in a conference which is "incomplete" in the sense that it stops after the `conf` collection and not taking, at least C_{23}, C_{24}, C_9 and C_{10} (the collections of `date` and `name` nodes, with their respectively leaf value collections).

6.4.2 Boundaries for DAG weights

For a main entity chosen based on its w_{DAG} score, its natural boundary is the DAG of the descendants of C^* restricted only to nodes which have contributed to the score of C^*. We denote those boundaries $bound_{DAG}$. For instance, Figure 6.8 shows $bound_{DAG}$ for the main entity `author`: only collections that contributed to the score of the `author` collection are in the boundary. Collections that are descendants of the `author` collection but did not transmit any weight to it because they are involved in cycles are not part of the boundary. However, w_{DAG} is able to identify deep boundaries, on the contrary to boundaries computed for simple scores. For example, the `author` collection obtains a boundary composed of C_9, C_{10} and all collections from C_{15} to C_{22}. The affiliation depth would therefore be reflected. However, note that, because w_{DAG} does not cross cyclic

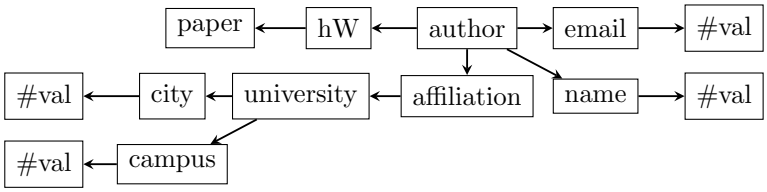


Figure 6.8: The $bound_{DAG}$ boundary obtained for the main entity `author`, based on w_{DAG} scores.

edges, $bound_{DAG}$ does not involve any collections included in a cycle.

6.4.3 Boundaries for PageRank-based weights

When using w_{PR} and w_{dw-PR}, no clear boundary may be defined based on the scores. Indeed, scores have been propagated globally and iteratively until convergence. Depending on the graph connectivity, all nodes may have contributed to all other nodes in the graph. Further, the $bound_{DAG}$ boundary method does not reflect the weight transfers across cyclic paths, thus it is not consistent with the PageRank propagation principle. Instead, we devise two flooding-style boundary computation methods, which leverage edges along the paths rooted in C^*. The idea is to go as deep as possible until finding a collection edge which *should not* be part of the boundary.

Flooding boundary

In the database literature [145], when an entity has a one-to-one correspondence with another, they are typically associated, or modeled together. For instance, if each person has at most an email, that email is considered part of the person. Including only such attributes in the boundary may be too restrictive, especially if few data nodes break the at-most-one assertion. Therefore, a second criterion for the flooding is to accept edges having an edge transfer factor above a given threshold, denoted f_{min}. For instance, even if some people may have several email addresses, the `email` collection would still be part of the boundary of `people` because the edge transfer factor between people and emails is 1, meaning that all emails belong to people (the collection is not shared). We define:

Definition 6.4.1 (*Flooding boundary*)

The **flooding boundary**, denoted $bound_{fl}$, associated to a collection C^* is the (sub-)graph rooted in C^* and encompassing each collection node $C_i \in C$ such that:

1. There exists at least one path starting in C^* and ending in C_i;
2. Among those, each collection edge along at least one such path meets at least one of the two following conditions:
   - It is an at-most-one collection edge;
   - Its edge transfer factor is above f_{min}.

Note that the boundary is defined on the original (not the reverse) collection graph $\mathcal{G}$.

As an example, the flooding boundary of the `author` main entity corresponds to the collection graph itself, i.e., it encompasses all collection nodes and edges of $\mathcal{G}$. Indeed, all collection edges are either

at-most-one (in Figure 5.4, the straight ones are) or have an edge transfer factor above f_{min}, which we set to 0.8 in our experiments (see Section 6.9).

We make a restriction only in the particular case of collection graphs produced out of XML documents, we limit the flooding boundary to collection edges that do *not* represent ID-IDREF data edges. This allows to not inline (include) the referenced elements into their parents, which may end encompassing the collection graph entirely.

Acyclic flooding boundary

When computed on a collection graph having cycles, the flooding boundary may accept all nodes and edges, when everything is reachable from C^* due to the cycles. In turn, this may include the whole collection graph in a single entity, which is not desirable.

Therefore, we define $bound_{fl-ac}$, the **acyclic flooding boundary**, an improved version of the flooding boundary which does not traverse in-cycle edges, as follows:

Definition 6.4.2 (*Acyclic flooding boundary*)

The **acyclic flooding boundary**, denoted $bound_{fl-ac}$, associated to C^* is the (sub-)graph rooted in C^* and encompassing each collection node $C_i \in C$ such that:

1. There exists at least one path starting in C^* and ending in C_i;
2. Among those, at least one of such path does not contain any in-cycle edge;
3. Among those acyclic paths, each collection edge along at least one such path meets at least one of the two following conditions:
   - It is an at-most-one collection edge;
   - Its edge transfer factor is above f_{min}.

Similarly, for collection graphs produced out of XML documents, the acyclic flooding boundary does not traverse collection edges corresponding to ID-IDREF data edges. The acyclic flooding boundary of the `author` collection corresponds to the blue area in Figure 6.1, including the author names, emails and deep affiliations, as well as their respective values. The collection C_{12} is not included in C_{14}'s boundary because it is part of the cycles between papers, authors, and conferences.

Alternative: Data-acyclic flooding boundary

The method $bound_{fl-ac}$ may lead to non-intuitive abstractions because the flooding stops as soon as it encounters a cycle in $\mathcal{G}$. For instance, Figures 5.6 and 5.7 show that an acyclic graph obtained from an XML document may lead to a cyclic collection graph. Next, say the scoring method selects the `email` collection. When its boundary is computed with the acyclic flooding boundary, it stops when the collection edge `ul` $\rightarrow$ `li` is encountered (because it is involved in a cycle). Therefore, the boundary of the collection `email` is clearly missing the collections `ul`, `li` and (li) `#val`, which we would expect it to be included.

The data-acyclic flooding boundary is based on the following two observations:

- When a cycle in $\mathcal{G}$ is also present in G, i.e., the corresponding data edges are also involved in a cycle, the node may belong to the main entity boundary, or to another main entity boundary.

- However, when a cycle in $\mathcal{G}$ is not present in G, it seems natural that the corresponding data should be part of the boundary. Conceptually, if there is no cycle at the data level, there is no doubt that it should belong to the boundary.

We define the data-acyclic flooding boundary as follows:

Definition 6.4.3 (*Data-acyclic flooding boundary*)

The **data-acyclic flooding boundary**, denoted $bound_{fl-dac}$, is the (sub-)graph rooted in C^* and encompassing each collection node $C_i \in C$ such that:

1. There exists at least one path starting in C^* and ending in C_i;
2. Among those, at least one path has all its in-cycle edges data-acyclic (if there are some);
3. Among the data-acyclic ones, each collection edge along at least one such path meets at least one of the two following conditions:
   - It is an at-most-one collection edge;
   - Its edge transfer factor is above f_{min}.

6.5 Collection graph update methods

We now present how to reflect, during the main entity selection algorithms, that some main entities have already been selected, together with their boundaries.

We define two **graph update methods**; the first one, presented in Section 6.5.1, is to be used together with simple scores, the other one, presented in Section 6.5.2, is to be used with more elaborate scores (recall data-weight DAG propagation and PageRank-based scores in Sections 6.3.2 and 6.3.3). By design, the naive and multi-greedy algorithms do not need a graph update step.

6.5.1 Graph update for simple scores

When using a simple scoring method, i.e., $desc_k$ or $leaf_k$, the graph update excludes from the collection graph all the collections that are involved in the boundary of an already-selected main entity. Any collection edge having its source or target collection excluded is also excluded from the collection graph. We call this method *boolean graph update* and denote it $update_{boolean}$.

While simple, this method has a major drawback. When shared collections (whether they are shallow or deeply shared, recall Section 5.3.1) that are part of a main entity boundary are excluded from the collection graph, this prevents other collections, that also point to those collections, from being included in other main entities' boundaries or to be reported as main entities. For instance, in Figure 6.1, if the greedy algorithm selects C_8, the `conf` collection, as a main entity, the collection C_9 (among others) will be excluded from the collection graph. Next, when C_{14} (`author` collection) is selected as a main entity, it no longer has the information that author nodes have names: the `name` collection cannot be included in the `author` boundary as it has been previously excluded.

6.5.2 Graph update for weight-based scores

For methods leveraging the data-weight DAG propagation or the PageRank-based scores, where importance is propagated across the graph, a simple boolean graph update is too limited. Instead, we devise a *data-based update* mechanism, which, once a main entity is selected, sets as *inactive* all

the *data* nodes and edges that are represented in C^*'s boundary, if they do not contribute to the weight of nodes outside the boundary.

We call this mechanism *exact graph update* and denote it a $update_{exact}$. Note that it is much more expensive, given that it needs to examine the whole data graph. Algorithm 9 details it. We inactivate each data node in the main entity (root) collection C^*(Line 2), and do the same for all its outgoing edges (Line 4). Next, we call a recursive, breadth-first inactivation, presented in Algorithm 10, on the target node of the inactive edge (Line 5). The recursive update takes as input a data node and decides whether it can be inactivated by checking whether all its incoming edges are inactive (Lines 2-5). If so, the node, and each of its outgoing edges, become inactive, then recursive calls are made on the targets of these edges. The algorithm continues until all data nodes and edges in the boundary have been traversed. Finally, the graph update recomputes a new collection graph, based on the remaining active collection nodes and edges, i.e., those representing at least one active data node, respectively data edge.

Algorithm 9: $update_{exact}$

```
   Input: G: a graph
1  foreach n ∈ C* do
2      n.isActive ← false
3      foreach edge e of the form n → t do
4          e.isActive ← false
5          recursiveUpdate(t)
6  G ← build()
```

Algorithm 10: Recursive node update

```
   Input: n: a data node
1  toInactivate ← true
2  foreach edge e of the form s → n do
3      if e is active then
4          toInactivate ← false
5          break
6  if toDeactivate then
7      n.isActive ← false
8      foreach edge e of the form n → t do
9          e.isActive ← false
10         recursiveUpdate(t)
```

The exact update could be implemented directly on the stored data, in our case with SQL queries, or in memory. The former does not require to load the data graph in memory, but suffers from low speed because, in general, recursive SQL queries are not executed efficiently. The latter is faster, but requires loading the data graph in memory, which may fail for large graphs. We implemented both alternatives; in this thesis, we report abstraction times measured when using the in-memory implementation.

We have also introduced two parameters that allow to *partially* load the graph in memory, together with a variant of the exact update implementation, based on this partial loading. This may yield a slightly different set of main entities being selected, due to the variant's not accessing the graph completely and simultaneously during updates. Thus, we do not consider this variant further.

6.6 Relationships between main entities

Once the main entities in the collection graph have been identified, we are interested in computing the set $\mathcal{MR}$ of their relationships. We first describe in Section 6.6.1 the algorithm to infer them from $\mathcal{ME}$, the set of main entities, and the collection graph. Next, in Section 6.6.2, we discuss how some relationships may contain collections that are worth reporting as main entities.

6.6.1 Relationships identification

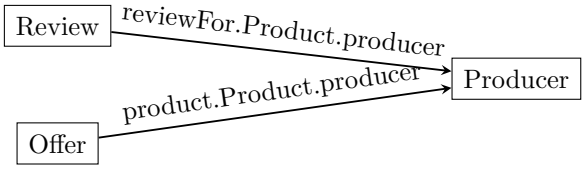
For each pair of main entity roots (C_i^*, C_j^*) such that there exists a set of paths from C_i^* to C_j^* in the collection graph, we report this set as the set of relationships between C_i^* and C_j^*. Indeed, two main entities may have several relationships, e.g., some authors write papers, which are themselves written by those authors, as in Figure 5.4. Each single relationship from C_i^* to C_j^* can be labeled with the corresponding path label (as defined in Section 5.3.3). For instance, in Figure 1.2, we have the path `person.watches.watch.watch@id.open_auction` between the main entities of `person` and `open_auction`, thus a relationship. In general, their number is moderate if E_{max} is low, as outlined in the experimental evaluation (see Section 6.9).

6.6.2 Multi-traversed (non-main) entities

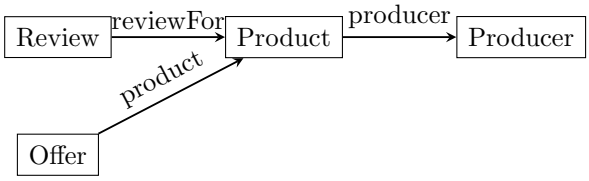
When an abstraction contains several main entities connected by multiple relationships, some (non-main) collections may be utilized in several relationships. We call such collections **multi-traversed** collections. This is, for instance, the case in Figure 6.9, which shows an abstraction built out of an e-commerce dataset where a set of products is manufactured by different producers and sold through offers; consumers may leave reviews on them. The greedy main entity selection algorithm has selected three main entities, those of reviews, offers and producers. Next, two relationships have been identified, i.e., `reviewFor.Product.producer` and `product.Product.producer`.

Among the multi-traversed collections, some deserve to be elected as main entities because they help the overall understanding of the abstraction and may simplify complex relationships traversing several collections. This is the case of the `Product` collection in Figure 6.9a. Relying on [153], we study how multi-traversed collections should be processed conceptually. When a collection C_j is pointed by only one collection C_i, C_j "belongs to" C_i, thus is *inlined* in C_i. Otherwise, when several collections are pointing C_j, C_j may be viewed as an entity of its own and should deserve to be elected as a main entity. Therefore, as a post-processing heuristic, the eligible non-main, multi-traversed collections are promoted to be shown as entities in the abstraction (not just on relationship labels). For instance, in Figure 6.9b, the collection of `Product` nodes has been promoted; `reviewFor`, `product` and `producer` are not eligible.

We now explain when such multi-traversed (non-main) entities exist. In our example, products are



(a) Three elected main entities with their relationships in a dataset about e-commerce.



(b) The abstraction after the multi-traversed entity is promoted as a main one.

Figure 6.9: Illustration of an abstraction before and after the election of multi-traversed non-main entities as main ones.

not elected because each single product is involved in an offer or a review. Thus, the algorithm elects offers and reviews first as they have a larger score. Next, the graph update makes inactive the collection of products as all products have been individually inactivated. Further, the collection of producers is reported because it still has some weight due to some producers that are not referenced in any product. Finally, the relationships are computed and the post-processing heuristic identifies the collection of products as a main entity. Therefore, it is promoted as a main entity and relationships are re-computed. More generally, consider two (or more) paths cp_i, cp_j of the form $C_{i_1} \rightarrow ... \rightarrow C_{i_m}$ and $C_{j_1} \rightarrow ... \rightarrow C_{j_n}$ connecting main entity root collections at the paths' extremities. No restriction about distinctness applies to collections in cp_i, respectively cp_j, e.g., C_{i_1} may equal C_{j_1}, and so on. Therefore, one, two, three or four main entities may be connected by these two paths. In Figure 6.9a, we have $C_{i_1} =$ `Review`, $C_{j_1} =$ `Offer` and $C_{i_m} = C_{j_n} =$ `Producer`. A (non-main) collection C_z that is present both in cp_i and cp_j exists if the following two conditions are true:

- All nodes in C_z are only pointed by nodes present in C_{i_1} and C_{j_1};
- C_{i_m} and/or C_{j_n} are still active; this happens in two cases:
  - They have nodes pointed by nodes in other collections, which are not in cp_i nor cp_j;
  - Or they have nodes not pointed by nodes in C_{i_1} nor C_{j_1}.

6.7 Alternative: multi-greedy algorithm

In Section 6.2, we presented two algorithms, the naive and greedy ones, which both leverage the *multi-traversed* collections post-processing heuristic (recall Section 6.6.2). Those collections do not have the chance to be elected because they are totally included in the boundary of their ancestors, something the heuristic lifts by reporting them. Yet, it may elect multi-traversed collections even though their score were low compared to others, thus decreasing their legitimacy regarding other selected collections. Also, in case of data-weight and PageRank-based scores, a main entity may have been chosen due to this multi-traversed collection, which increased its score. If the heuristic further promotes that multi-traversed collection, the reason for choosing that main entity becomes weaker.

To lift the above-mentioned limitations, we introduce an alternative algorithm which selects several main entities at a time. We call it *multi-greedy* algorithm and present it in Algorithm 11. The idea

is to select, at each iteration of the greedy, a pool *tops*, of size *top*, of eligible collections maximizing the current scores (Lines 6-13). Selecting several, i.e., *top*, main entities at a time allows to elect the best-scored collections, even though they were fully included in their ancestors. Next, if the pool is empty, meaning that no collection is eligible anymore, the algorithm stops. Otherwise, for each collection in the pool, its root is set, and its boundary is computed (Lines 17-19). In this algorithm, we do not need a graph update because several collections are selected at a time to avoid inlining too many collections, something that the exact graph update was made for. The algorithm stops when E_{max} main entities have been elected, or when a sufficient data coverage is achieved.

Algorithm 11: Multi-greedy main entity selection algorithm

Input: $\mathcal{G}$: a collection graph, *top*: a positive integer
Output: $\mathcal{ME}$: the set of main entities found in $\mathcal{G}$

```
1  cws ← [ ]
2  tops ← ∅
3  for C_i ∈ C do
4      cws[C_i] ← score(C_i)
5  while |ME| < E_max or cov < cov_min do
6      i ← 0
7      for i ≤ top do
8          C* ← nextCollection(cws)
9          if C* = −1 then
10             stop
11         else
12             tops ← tops ∪ { C* }
13             i ← i + 1
14     if |tops| = 0 then
15         stop
16     else
17         for t ∈ tops do
18             b ← boundary(t)
19             ME ← ME ∪ {(t, b)}
```

When top = 1, Algorithm 11 behaves like the naive one (Algorithm 2). **When top =** E_{max}, Algorithm 11 selects directly the final set of main entities. In this case, shared collections are not inlined in other main entities' boundaries; instead, they are all elected as main entities, which lifts the problem that the had-hoc heuristic for multi-traversed entities was trying to solve. **When (**E_{max} mod **top)** $>$ **0**, a pool of some main entities will be greedily reported, until E_{max} is reached. Note that we may report some more collections that what specified with E_{max}, specifically E_{max} mod *top* additional collections.

6.8 Main entity classification

Now that we have selected a few main entities and identified their relationships, we aim at **classifying each main entity among a set of semantic classes**, or categories. Indeed, they may have user-friendly labels, such as `conf`, `paper` and `author` in Figure 6.1, but this is not always the case as shown in our experimental evaluation in Section 6.9.

To assign a semantic class to a main entity $\mathcal{ME}_i \in \mathcal{ME}$, we describe in Section 6.8.1 a set of *semantic resources* as well as *embeddings* to assign to $\mathcal{ME}_i$ a realistic and easy-to-understand category. Next, in Section 6.8.2, we detail our classification algorithm, which assigns a semantic class to each main entity. Finally, we discuss in Section 6.8.3 alternatives to further add in the classification algorithm.

6.8.1 Semantic resources

We define a set of categories $\mathcal{K}$, e.g., `Person`, `GeographicalLocation`, `Restaurant`, and a set of *properties* $\mathcal{P}$ that are associated to categories as follows:

- For each property $p \in \mathcal{P}$, the set $domain(p) \in \mathcal{K}$ describes categories to which one belong if one has the property p. For instance, $domain(birthDate) = \{Person\}$ states that if one has a property `birthDate`, it may be classified as a `Person`.
- Similarly, the set $range(p) \in \mathcal{K}$ describes categories to which the values of p belong; for instance, $range(birthDate) = \{Date\}$.

Defining few core concepts and their associated properties

To populate $\mathcal{K}$ and $\mathcal{P}$, users may have in mind few (generic) concepts/categories, e.g., `Person`, `Location`, `Organization`, and some properties associated to them, e.g., `name`, `address` and `age` for `Person`. We manually identified **seven categories** and few corresponding properties:

- `Person`, to which we have associated 40 properties including `name`, `address`, `age`, `worksFor`;
- `Location`, to which we have associated 31 properties including `language`, `latitude`, `landscape`;
- `Organization`, to which we have associated 30 properties including `name`, `logo`, `rating`;
- `Creative Work`, to which we have associated 20 properties including `name`, `rating`, `publisher`;
- `Event`, to which we have associated 23 properties including `name`, `organizer`, `location`;
- `Product`, to which we have associated 29 properties including `name`, `quantity`, `price`;
- `Thing`, which is a general category that we will assign to main entities that do not fall in one of the six specific categories described above. No specific properties are associated to the `Thing` category as this is a default category.

Note how some properties belong to several categories, such as `name` and `rating`. In total, we obtained 101 distinct properties by looking at Wikidata, Schema.org and XMLNS FOAF. We denote by $\mathcal{P}_k$ the set of properties associated to the category $k \in \mathcal{K}$.

```
<yago:EmmanuelMacron> <rdf:type> <yago:Person> .
<yago:EmmanuelMacron> <yago:nationality> "French" .
<yago:EmmanuelMacron> <yago:spouse> <yago:BrigtitteMacron> .
```

Figure 6.10: A set of RDF triples describing the RDF entity Emmanuel Macron.

Extending core concepts and properties with knowledge bases

The manual identification of categories and properties leads to a very tiny set for both, which is not sufficient to correctly classify most of the main entities. Therefore, we leverage knowledge bases (recall Section 2.6) to augment them in terms of quantity (number of categories and properties) and quality (a large number of properties per category).

First, we need to **anchor our core concepts to KB classes**. Given a knowledge base B, we map each core concept to a few classes from B, using word distance, e.g., Word2Vec [125]. For instance, our category `Location` is mapped to `http://dbpedia.org/ontology/Place` and `schema:Place`, respectively from the DBPedia [18] and YAGO4 [138] knowledge bases. We add to $\mathcal{K}$ the B classes thus obtained.

Next, we need to **acquire properties for categories in $\mathcal{K}$ using KBs**. Given B and a category $k \in \mathcal{K}$, the set of properties $\mathcal{P}_k$ likely to be associated with k is obtained using Equation 6.1.

$$\mathcal{P}_k = \{r \mid \langle \texttt{a} \rangle \langle \texttt{r} \rangle \langle \texttt{b} \rangle \in B \wedge \langle \texttt{a} \rangle \langle \texttt{rdf:type} \rangle \langle k \rangle \in B\} \tag{6.1}$$

For example, triples in Figure 6.10, which have been extracted from YAGO4 [138], lead to the addition of `nationality` and `spouse` to the set $\mathcal{P}_{Person}$.

Further, to continue expending the set of categories $\mathcal{K}$ and their associated properties, we **acquire $\mathcal{K}$ categories and $\mathcal{P}$ properties from GitTables [94]**. GitTables is a repository of 1.5M tables extracted from CSV files in GitHub. GitTables has been populated using SHERLOCK [95], a state-of-the-art deep-learning based semantic tag annotation technique for a collection of values. For each attribute name encountered in a table, it provides candidate properties from DBPedia [18] and/or Schema.org. For instance, the attribute name "`impact factor`" is associated to the DBPedia property `http://dbpedia.org/ontology/impactFactor`, while "`actor`" is associated to the Schema.org `schema:actor` property. GitTables also provides the domain and range sets corresponding to these properties. Each property in GitTables is a candidate to be added in $\mathcal{P}$; similarly, each category in the domain of a GitTables property is a candidate to be added in $\mathcal{K}$. For instance, Figure 6.11 shows the GitTable's entry for `gender`. Therefore, `schema:SportsTeam` would be added to $\mathcal{K}$ (`schema:Person` is already in $\mathcal{K}$, following the anchoring of our core concepts in KB classes); `gender` would be added to $\mathcal{P}_{Person}$ and $\mathcal{P}_{schema:SportsTeam}$.

Dealing with worthless properties

The acquisition of properties using knowledge bases and GitTables may retrieve properties containing inaccurate information or being very general: we call them worthless properties. For example, an erroneous information is represented by the triples $\langle$`Emmanuel Macron`$\rangle\langle$`rdf:type`$\rangle\langle$`Organization`$\rangle$ and $\langle$`Emmanuel Macron`$\rangle\langle$`birthdate`$\rangle\langle$`1977`$\rangle$. Indeed, `Organization` resources do not have a `birthdate`.

```
{
        "id": "schema:gender",
        "label": "gender",
        "description": "Gender of something, typically a Person",
        "domain": ["schema:Person", "schema:SportsTeam"],
        "range": ["schema:GenderType", "schema:Text"]
}
```

Figure 6.11: The GitTable entry for the property `gender`.

This happens because knowledge bases such as Wikidata are collaboratively created, which can lead to errors. Moreover, we might get some properties which are common for all the categories in $\mathcal{K}$, thus are not very useful to classify a main entity in a unique category. For example, the property `Google Knowledge Graph ID` has been retrieved for each core category in $\mathcal{K}$. Therefore, we **score retrieved properties and rank them to select only the ones with the highest scores.**

We define the **category score**, measuring how frequent a property is in a given category, as follows:

Definition 6.8.1 (*Category score*)

Given $\mathcal{P}_k$ the set of properties retrieved for a category $k \in \mathcal{K}$ from a knowledge base B, the **category score** of a property p for a category k is:

$$c\_score(p, k) = |\{a | \langle \texttt{a} \rangle \langle p \rangle \langle \texttt{b} \rangle \in KB \wedge \langle \texttt{a} \rangle \langle \texttt{rdf:type} \rangle \langle k \rangle \in B\}|$$

such that $0 \leq c\_score(p, k) \leq \infty$.

Intuitively, a property which appears only few times in B is less likely to be relevant than a property which appears very often. For example, the pair (`birthdate`, `Organization`) obtains a low c_score since only a few `Organization` resources have a `birthdate` in B. Inversely, a property such as `ISBN` will have a high c_score since almost each `Organization` has an ISBN number.

Next, we look for very general properties by introducing the **inverse category score**: this quantifies how unique a property is for distinguishing categories. Formally, we have:

Definition 6.8.2 (*Inverse category score*)

Given $\mathcal{P}_{all} = \cup_{k \in \mathcal{K}} \{\mathcal{P}_k\}$ the set of set of properties retrieved for each category $k \in \mathcal{K}$ according to Equation 6.1, the **inverse category score** of a property p is:

$$ivc\_score(p) = \log\left(\frac{1 + |\mathcal{K}|}{1 + |\{\mathcal{P}_k \,|\, p \in \mathcal{P}_k, \mathcal{P}_k \in \mathcal{P}_{all}\}|}\right)$$

such that $0 \leq ivc\_score(p) \leq \infty$.

Concretely, a property p appearing in all sets $\mathcal{P}_k \in \mathcal{P}_{all}$ will get $ivc\_score(p) = 0$ because $\log(1) = 0$. A property p appearing in only one $\mathcal{P}_k$ will get a high score. For example, the property `Google Knowledge Graph ID` gets a low score because it participates to all categories in $\mathcal{K}$. Inversely, `ISBN` gets a high inverse category score because it appears only for Organization resources.

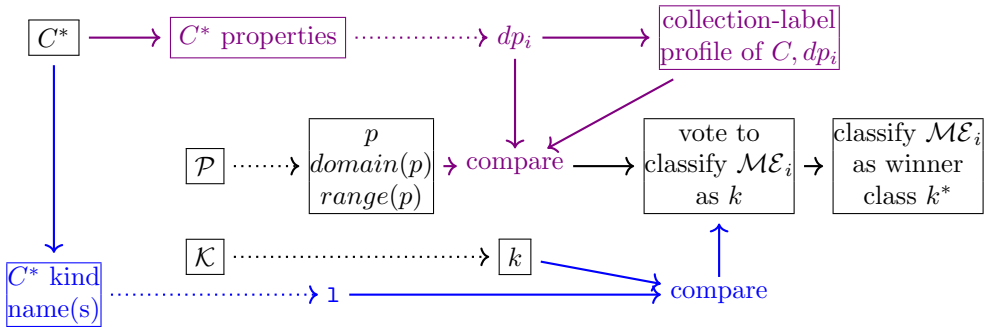


Figure 6.12: Outline of the classification algorithm.

Finally, after defining the category score *c_score* and the inverse category score *ivc_score*, we define the score of a property p for a category k as the product of the category score and the inverse category score:

Definition 6.8.3 (*Property score*)

Given a property p, and a category $k \in \mathcal{K}$, the **property score** of p in k is:

$$score(p, k) = c\_score(p, k) * ivc\_score(p)$$

For each category $k \in \mathcal{K}$, we score the properties in decreasing order of *score* and retain the top-m $\mathcal{P}_k^m$ properties. For each $p \in \mathcal{P}_k^m$, we set its domain as being $\{k\}$ and its range as being $range(p)$, if the range is available from B, $\emptyset$ else. Each such property is added to $\mathcal{P}$ and categories in the domain of p that are not yet in $\mathcal{K}$ are added to it.

6.8.2 Classification algorithm

Algorithm 12 details the classification of a main entity $\mathcal{ME}_i$ in a category from $\mathcal{K}$. Its main steps are also outlined in Figure 6.12; solid arrows connect associated data items and trace the classification process, while dotted arrows go from a set to one of its elements.

To assign a semantic class to a main entity $\mathcal{ME}_i \in \mathcal{ME}$, we leverage four different signals:

1. The main entity root kind name(s), if available (recall Section 5.1);
2. The $\mathcal{ME}_i$'s records labels;
3. The labels of first-level collections in $\mathcal{ME}_i$'s boundary - this corresponds to the main entity root children;
4. The collection-label entity profiles (recall Section 5.3.1) associated to each pair of a main entity root and a main entity root child label - this corresponds to the Named Entities extracted from the values of these root children.

First and foremost, we initialize $S[k]$ to 0 for each category $k \in \mathcal{K}$. This will store the classification score that each category obtains for the given main entity (Lines 1-2). Next, if $\mathcal{ME}_i$ has one or a few kind names, we compute the semantic similarity of each pair of a kind name kn and a category k (Lines 3-5) using Equation 6.2. This basically computes the similarity between two strings, by embedding each keyword w they contain in a multidimensional vector $e(w)$, and summing up the pairwise cosine similarities of these embedding vectors. This rewards semantic similarity, e.g., `anniversary` is considered similar to `birth date`, even if they are different words. We currently use the Word2Vec model [125] for this task. The similarity scores thus obtained are added to $S[k]$. This leverages signal (1). Further, we compute the word similarity between each main entity record label and the categories in $\mathcal{K}$ to leverage signal (2) (Lines 6-7). Recall that some main entities have records with all the same label, e.g., XML elements, while others have records having each a distinct label, e.g., RDF resources. Third, we leverage signals (3) and (4). For this, we determine the set dp of *data properties of* C^*, i.e., the set of collection labels outgoing the main entity root. We compare each data property $dp_i \in dp$, with each semantic property $p \in \mathcal{P}$, and assign to each pair a similarity score. This score, denoted $score(dp_i, p)$ and defined in Equation 6.4, reflects:

- The similarity between the names of the properties dp_i and p, which is computed using the word similarity of Equation 6.2;
- Which we possibly average (if there are extracted entities in the leaf values of the data property dp_i) with the entity similarity computed using the Named Entity types found in the values of dp_i (represented in collection-label entity profiles) and the categories in $range(p)$ using Equation 6.3. This computes entity type overlap as a ratio of the total length of the Named Entities in the values of dp_i to the total length of values of dp_i.

$$w\_sim(s_1, s_2) = \sum_{w_i \in s_1, w_j \in s_2} cosine\_sim(e(w_i), e(w_j)) \tag{6.2}$$

$$entity\_sim(C_i, dp_i, p) = \frac{\sum_{k \in \tau_{C_i,dp_i}.l, k \in range(p)} (\tau_{C_i,dp_i}.l[k])}{|range(p)| \cdot \tau_{C_i,dp_i}.s} \tag{6.3}$$

$$score(C_i, dp_i, p) = \begin{cases} \frac{w\_sim(dp_i,p) + entity\_sim(C_i,dp_i,p)}{2} & \tau_{C_i,dp_i}.l > 0 \\ w\_sim(dp_i, p) & \text{otherwise} \end{cases} \tag{6.4}$$

We retain p as a match of dp_i if the similarity score, presented in Equation 6.4, is above a threshold α (Lines 8-13). Next, each retained property p "votes" toward classifying $\mathcal{ME}_i$ in any category $k \in domain(p)$, by contributing to $S[k]$ according to Equation 6.5 (Lines 15-16).

Intuitively, this *transfers the domain knowledge about* $\mathcal{P}$*, our semantic properties* (recall semantic resources presented in Section 6.8.1), to the data properties of $\mathcal{ME}_i$, in order to apply a soft (quantitative) domain reasoning. This mimics the "hard" (logical) reasoning based on RDFS domain constraints ("if x has a property p and $\tau \in domain(p)$, then x is of type τ"), but we modulate the strength of each property's vote, as follows:

- The higher $score(C_i, dp_i, p)$ is, the more the domain constraint(s) of p matter, because it is more likely that p and dp_i mean the same or similar things;

- p "splits its vote" equally among all classes in $domain(p)$, to reward properties with few domain that type more precisely;
- p's votes are multiplied by the node support, i.e., the percentage of nodes in $\mathcal{ME}_i$ having property dp_i. The intuition is that the more frequent dp_i is, the more the domain typing it brings.

$$\frac{score(C_i, dp_i, p) \times supp(dp_i, C_i)}{|domain(p)|} \tag{6.5}$$

Based on the computed scores $S[k]$, we classify $\mathcal{ME}_i$ with the highest-scored category $k \in \mathcal{K}$, if that is well-defined (Lines 19-33). First, we look for the categories achieving the highest score max (Lines 19-21). If only one category has this score, thus *best* contains only one element, we classify $\mathcal{ME}_i$ with that category. Otherwise, there exist ties and we have to solve them using heuristics. We proceed as follows:

- First, we check whether there exists a hierarchy between the categories involved in the tie: if so, we classify $\mathcal{ME}_i$ using the most specialized category. For instance, if there is a tie between the categories `schema:Hospital` and `schema:MedicalOrganization` and `schema:Hospital` is a sub-class of `schema:MedicalOrganization`, we would classify the main entity as an hospital.
- Otherwise, if the main entity root C_i has a data property labeled with *labels typically associated to identifiers*, e.g., `id`, `label`, `name` or `designation`, and if all the values of this property contain only Named Entities mapped to a single $\mathcal{K}$ category, we classify $\mathcal{ME}_i$ with that category.

If the above heuristics do not suffice to classify $\mathcal{ME}_i$ in a single $\mathcal{K}$ category, we assign it the most general category in $\mathcal{K}$, e.g., `Thing` in our setting.

6.8.3 Alternatives

As of now, the classification algorithm, presented in Algorithm 12, leverages the set of labels of collections outgoing the main entity root. When the main entity boundary is complex, e.g., involves collections nested several levels deep, the information brought by grandchildren and their descendants is not used. This means that when a data property does not have a leaf collection as a child, the associated entity profile is null, thus that data property is compared to semantic properties only using the word similarity (recall Equation 6.4). This may prevent good classification results for deep main entities. For instance, in Figure 6.1, the classification algorithm will classify the collection of authors knowing that they have a `name`, an `email` and an `affiliation`. However, it would not know that the data property `affiliation` leads to Organization entities (because universities can be classified so using their own data properties and collection-label entity profiles). Therefore, an alternative is to **leverage the boundary deep structure of data properties**. This would induce a depth-first traversal of the main entity boundary to classify the main entity using:

- First, its direct non-deep children, i.e., its data properties and their associated collection-label entity profiles;

Algorithm 12: Classifying a collection C_i.

Input: $\mathcal{ME}_i$: a main entity, C_i: $\mathcal{ME}_i$'s root, $\mathcal{P}$: semantic properties, $\mathcal{K}$: categories
Output: $\mathcal{ME}_i$: classified main entity

foreach $k \in \mathcal{K}$ **do**
  $S[k] \leftarrow 0$
/* 1. compare $\mathcal{ME}_i$'s kind names and $\mathcal{K}$ categories */
if C_i *has one or a few kind names* kns **then**
  foreach $k \in \mathcal{K}, kn \in kns$ **do**
    $S[k] \leftarrow S[k] + w\_sim(kn, k)$
/* 2. compare $\mathcal{ME}_i$'s records labels and $\mathcal{K}$ categories */
foreach $k \in \mathcal{K}, l \in \mathcal{L}$ **do**
  $S[k] \leftarrow S[k] + w\_sim(l, k)$
/* 3. compare $\mathcal{ME}_i$'s outgoing properties names and $\mathcal{K}$ categories */
foreach dp_i *property of some nodes in* C_i **do**
  foreach $p \in \mathcal{P}$ **do**
    if $\tau_{C_i,dp_i}.l > 0$ **then**
      $score(dp_i, p) \leftarrow \frac{w\_sim(dp_i,p)+entity\_sim(C_i,dp_i,p)}{2}$
    else
      $score(dp_i, p) \leftarrow w\_sim(dp_i, p)$
    if $score(dp_i, p) > \alpha$ **then**
      foreach $k \in domain(p)$ **do**
        $S[k] \leftarrow S[k] + \frac{score(C_i,dp_i,p)\cdot supp(dp_i,c)}{|domain(p)|}$
    else
      $score(dp_i, p) = 0$
/* 4. select the best category for $\mathcal{ME}_i$ */
$max \leftarrow max_k\{S[k]\}$
if $max > \theta$ **then**
  $best \leftarrow argmax\_all(S)$
  if *best has just one element* k^* **then**
    Classify $\mathcal{ME}_i$ as k^*
  else
    if $\exists\, \overline{k} \in$ *best s.t.* $\overline{k}$ *is a specialization of all the other best categories* **then**
      Classify $\mathcal{ME}_i$ as $\overline{k}$
    else
      if C_i *has an ID-like and its entity profile contains 1 type* **then**
        Classify $\mathcal{ME}_i$ as that type
      else
        Classify $\mathcal{ME}_i$ as the most general class in $\mathcal{K}$
else
  Classify $\mathcal{ME}_i$ as the most general class in $\mathcal{K}$

- Then, its direct deep children, which would be classified first, and then the chosen category would correspond to a "kind of" collection-label entity profile saying that entities of that category have been found for that (deep) data property.

6.9 Experimental evaluation

We implemented data abstractions in ABSTRA, a Java software leveraging CONNECTIONLENS [13] for the graph creation (recall Chapter 4). We experimented on a Linux server with an Intel Xeon Gold 5218 CPU @ 2.30GHz and 196GB of RAM, and relied on PostgreSQL v9.6 for storing the data.

We focus our evaluation on **JSON**, **PG**, **RDF** and **XML** datasets, given their popularity, and because their complexity make their abstraction more challenging (Section 6.9.1). The questions we study are:

1. *What is the best-performing main entity selection method?* (Section 6.9.2);
2. *Does this method succeed in identifying the main entities, their boundaries, and the main relationships in a dataset?* (Section 6.9.3);
3. *How well does the classification algorithm perform?* (Section 6.9.4);
4. *How efficiently can abstractions be computed?* (Section 6.9.5);
5. *How do they compare to inferred schemas?* (Section 6.9.6);

Finally, we discuss interesting remarks we found on abstractions in Section 6.9.7.

6.9.1 Datasets, semantic resources, and settings

We evaluated ABSTRA on JSON, PG, RDF and XML synthetic and real-life, open datasets. Table 6.1 shows, for each dataset, the number of nodes and edges in G_0 (the graph obtained by loading the dataset) and the normalized graph G; $\diamond$ indicates synthetic datasets (as opposed to real-world). Since each RDF edge is labeled, normalization (Section 4.3) adds many extra nodes and edges.

For **JSON**, we wrote the Researchers dataset generator. Each researcher has a first and last name, id, H-index, status, gender, age, date of birth, and continent. It also includes: the titles of their three best papers, and a list of five co-authors (each with a first name and a last name). Real-life JSON datasets comprise a dataset of commits and push events in GitHub repositories [184], Prescriptions [192] (medical prescriptions), NYTimes articles [20], three datasets from Yelp [195], a crowd-source app to review businesses, and bibliographic notices from CoreResearch [181].

For **PG** datasets, we used two datasets generated based on the LDBC benchmark [64]: LDBCsmall, used in [34], and LDBC0.3 obtained using the available generator. We also wrote a custom generator to obtain Movies250K. Each movie has a title, a synopsis, a duration, a year and a number of entries. Some movies are also typed as sequel. Movie directors have a first and a last name while actors also have a nationality. Actors and movie directors are also both typed as Person in addition to their own type. Cinemas have a name and a city. Awards have a label, a year and a boolean indicating whether it is for international nominations. Each movie is projected in some cinemas at a specific

date, and some of them received awards. Each actor has played in some movie(s); for each acting, a role is associated. Finally, movie directors have supervised the production of one or several movies.

For **RDF**, we used BSBM [30] and LUBM [82] benchmarks, as well as our generator of graphs about scientific papers (having a title, a DOI and a year) written by authors (with first name, last name, affiliation university, birth date, gender, email and honorific prefix) and published in conferences (with a place, a year, an organizer, and a duration). Authors are also invited in conferences, thus leading to cycles. We used real-life datasets about: gas and electricity in Italy (EnelShops [182]), food recipes (Foodista [183]) and NASA flights [190].

For **XML**, we used XMark [150] documents, modeling an e-commerce scenario where people buy and sell on products through bids. We also used the real-life PubMed [193], Mondial [189] and WikiMedia [194] datasets.

For the classification (Section 6.8), we used YAGO4 [138] and WikiData as **knowledge bases**. We extracted 565 $\mathcal{P}$ properties (on top of the 101 we had manually identified). From GitTables [94], we derived 4.187 $\mathcal{P}$ properties; 3.687 (respectively, 3.898) among them have domain (respectively, range) statements, involving a total of 810 classes.

Settings Unless otherwise specified, we used the following values, which worked best in our experiments. For the main entity selection algorithms (recall Section 6.2), we set $E_{max} = 5$ to select at most 5 main entities and $cov_{min} = 1.0$ to get a full coverage of the dataset if possible. For the main entity classification (Section 6.8), $\theta = 0.8$ to assign a category to a main entity only if the score is sufficiently high and $\alpha = 0.3$ to use semantic properties that somehow match data properties.

6.9.2 Quality of the main entity selection methods

We assessed the quality of the main entities $\mathcal{ME}$ and relationships $\mathcal{MR}$ selected as described in Section 6.2 through a user study. For this task, we built four **dataset samples**, one for each data model, as subsets of datasets in Table 6.1. The samples are small enough (Table 6.2) for users to inspect them to decide if our abstractions are relevant, yet sufficiently complex to test our algorithms.

The **JSON sample** is extracted from the Researchers dataset. The **PG sample** is extracted from LDBCsmall. It contains forums, in which posts and comments are posted by people who study or work at an institution. Each forum, post or comment may have some tags. The **RDF sample** is a subset of Conferences. The **XML sample** is extracted from XMark1. It depicts items to be sold in auctions, people with nested addresses, some of which are interested in open auctions; there are also closed auctions. Items for sale have nested descriptions and categories they belong to. There is also a category hierarchy. As in larger datasets, items are in several continents, e.g., ⟨`africa`⟩ and ⟨`asia`⟩ XML elements.

We abstracted each sample using **11 methods**[2] (first column of Table 6.3), each of which pairs a scoring method (Sections 6.3.1, 6.3.2 and 6.3.3) with a method for determining its boundary (Sections 6.4.1, 6.4.2 and 6.4.3). The main entity selection algorithm used is the greedy one (Algorithm 4 in Section 6.2). We asked a group of **16 evaluators** (graduate and post-graduate researchers) to rank, for each sample, data abstractions shown as E-R diagrams, where they could click to unfold

[2]We did not experiment with the data-acyclic flooding boundary nor the multi-greedy main entity selection algorithm because they have been devised at the end of the PhD, after the experimental evaluation has been conducted.

Dataset name	$\|N_0\|$	$\|E_0\|$	$\|N\|$	$\|E\|$
JSON				
CoreResearch	2,112,023	2,112,022	2,112,025	2,112,024
GitHub	5,094	5,093	5,096	5,095
NYTimes	1,053,979	1,053,978	1,053,981	1,053,980
Prescriptions	14,214,033	14,214,032	14,214,035	14,214,034
Researchers ◇	4,505,657	4,505,656	4,505,659	4,505,658
YelpBusiness	8,111,982	8,111,981	8,111,984	8,111,983
YelpCheckIn	659,653	659,652	659,655	659,654
YelpTips	9,998,068	9,998,067	9,998,070	9,998,069
PG				
LDBCsmall ◇	9,851	43,011	48,621	77,546
LDBC0.3 ◇	2,306,540	10,907,534	11,890,404	19,167,734
Movies250K ◇	12,386,250	14,817,871	23,140,245	21,507,996
RDF				
BSBM4M ◇	1,134,834	4,000,740	4,755,392	7,241,116
BSBM16M ◇	5,344,748	16,038,464	19,870,744	29,051,992
Conferences ◇	121	184	273	304
EnelShops	14,694	51,639	56,426	83,464
Foodista	190,271	1,019,801	1,162,086	1,943,630
LUBM1M ◇	248,261	1,035,610	1,076,519	1,656,516
NASA	53,528	99,423	140,540	174,024
XML				
Mondial	129,567	135,588	130,756	136,777
PubMed	49,035	49,034	49,037	49,036
XMark1 ◇	3,392,392	3,392,391	3,392,394	3,392,393
XMark4 ◇	13,615,551	13,615,550	13,615,553	13,615,552
Wikimedia	1,824,185	1,768,803	1,872,877	1,817,495

Table 6.1: Datasets used in the evaluation.

Dataset name	$\|N_0\|$	$\|E_0\|$	$\|N\|$	$\|E\|$	$\|C\|$
Researchers (JSON)	569	665	684	780	26
LDBC (PG)	445	791	1,082	1,280	78
Conferences (RDF)	145	256	369	448	32
XMark (XML)	847	863	904	920	126

Table 6.2: Datasets used in the user relevance feedback.

nested attributes. We asked them to rank them according to **what a good abstraction is**, which leads to the following set of questions:

- *Does it capture all important entities? Are there spurious entities?*
- *Are the relationships informative? Do they help understand the entities?*
- *Do each entity's attributes logically belong there? Are any attributes missing?*

For each abstraction method and sample dataset, Table 6.3 reports the number of times that the method was ranked first, respectively in the top-3 methods. The ($leaf_1$, $bound_{leaf}$) is omitted, since it always leads to an empty abstraction: the only collections with a non-zero score (Section 6.3.1) are not eligible (recall eligibility criteria presented in Section 6.2). We show in bold the best-performing method(s) for each dataset.

On **JSON**, our weight-based methods perform (much) better than the baselines, except ($desc_3$, $bound_{desc}$) which coincides with the weighted methods. All weight-based abstractions of this dataset led to the same result. In the JSON sample abstraction, we fused the "co-authors" collection with the one of researchers because they both have similar properties (recall Section 5.2.3). On the **PG** sample, again all weight-based methods lead to the same result, and are ranked best; some simple baselines come close. They compute similar results as the weighted ones, but they either miss the `comment` entity (even though important) or report the `organization` entity, which can be nested into the `person` entity. On the **RDF** sample, users prefer some simple baselines to the weight-based methods. This is because all weight-based methods included `university` within the `person` entity, given that `university` only appears there; in contrast, users preferred to see `university` as a separate entity. The **XML** sample, with the most complex structure, lead to more diverse evaluations. (w_{PR}, $bound_{fl-ac}$) and (w_{dw-PR}, $bound_{fl-ac}$) win, followed by (w_{PR}, $bound_{fl}$) and (w_{dw-PR}, $bound_{fl}$). Users downgraded baseline methods, which fail to reflect the complex XMark structure.

In the Total column, ($leaf_2$, $bound_{leaf}$) is ranked 1st most often: this method only chooses tuple-like entities, with named properties having atomic values. The samples used here were simple, to ease user evaluation. In real-life datasets with hundreds of collections (see Tables 6.4, 6.5, 6.6 and 6.7), ($leaf_2$, $bound_{leaf}$) can only return E_{max} such entities, which degrades its coverage. The methods (w_{PR}, $bound_{fl-ac}$) and (w_{dw-PR}, $bound_{fl-ac}$) are close behind in the 1st place ranking, and they both score best in the top-3 ranking. Based on this, and the remark that only (w_{dw-PR}, $bound_{fl-ac}$) leverages both complex graph structure and data cardinalities, from now on, we focus on the **(w_{dw-PR}, $bound_{fl-ac}$) abstraction method, which performed best** in our experiments.

6.9.3 Main entities in all datasets

We now study the main entities $\mathcal{ME}$ and relationships $\mathcal{MR}$ identified by the (w_{dw-PR}, $bound_{fl-ac}$) abstraction method, in all our datasets (Tables 6.4, 6.5, 6.6 and 6.7). A ↻ indicates that the dataset's collection graph has cycles (14 datasets out of 23). For each dataset, we report $|C|$, the collection graph size; $|\mathcal{ME}|$, the number of selected main entities; $|\mathcal{MR}|$, the number of relationships connecting main entities; *cov* the data coverage of the abstraction; $\mathcal{ME}$, the set of main entities. For each main entity, we report:

1. A human-readable label (not yet our classification - see Section 6.9.4) obtained as follows. When the nodes in the collection share an explicit kind name, e.g., Post in LDBC data, we

Abstraction method	JSON		PG		RDF		XML		Total	
	1st	top-3	1st	top-3	1st	top-3	1st	top-3	1st	top-3
($desc_1$, $bound_{desc}$)	2	8	**8**	14	**16**	**16**	2	5	28	43
($desc_2$, $bound_{desc}$)	1	**15**	0	8	0	8	2	10	3	41
($desc_3$, $bound_{desc}$)	**11**	**15**	1	13	0	15	0	5	12	48
($leaf_2$, $bound_{leaf}$)	1	9	**8**	14	**16**	**16**	5	7	**30**	46
($leaf_3$, $bound_{leaf}$)	1	3	**8**	14	**16**	**16**	0	8	25	41
(w_{PR}, $bound_{fl}$)	**11**	**15**	**8**	**16**	0	1	6	11	25	43
(w_{PR}, $bound_{fl-ac}$)	**11**	**15**	**8**	**16**	0	12	**7**	**13**	26	**56**
(w_{dw-PR}, $bound_{fl}$)	**11**	**15**	**8**	**16**	0	1	6	11	25	43
(w_{dw-PR}, $bound_{fl-ac}$)	**11**	**15**	**8**	**16**	0	12	**7**	**13**	26	**56**

Table 6.3: Users' ranking of dataset sample abstractions.

show it in normal font. Otherwise, we provide ourselves a label, shown in *italic*, e.g., *Notice* in CoreResearch.

2. d_{max}, the *maximal depth in the data* of any record in the main entity root.
3. $|\mathcal{ME}_i|$, the number of nodes in the main entity root, i.e., its number of records.

Our JSON datasets lead to one entity and no relationships. This is because (*i*) unlike the sample used in Section 5.3, they feature no shared entities; (*ii*) they don't use references in the style of XML ID-IDREF. In the RDF BSBM4M dataset, the collection `Product` is multi-traversed (recall Section 6.6.2) by the relationships `Review.reviewFor.Product.productFeature.ProductFeature` and `Offer.product.Product.productFeature.ProductFeature`; this is why 6 main entities are reported.

Tables 6.4, 6.5, 6.6 and 6.7 show that **from each dataset, the selected entities are frequent, coherent, and semantically central**: this confirms that our approach, based on the collection graph and the main entity selection method (w_{dw-PR}, $bound_{fl-ac}$), attains its goal. The main entity depth varies from 2 (XMark) to 16 (LDBC0.3), with 3, 4 and 6 being frequent values. This shows that (w_{dw-PR}, $bound_{fl-ac}$) **identifies nested entities of various depths**, which fixed-depth baseline methods (Section 6.3.1) cannot do. Finally, the node coverage (*cov* in Tables 6.4, 6.5, 6.6 and 6.7) shows that, in general, **our abstractions represent most of the dataset**, thus fulfilling our objective (recall Section 6.1 presenting data abstraction requirements).

6.9.4 Quality of main entity classification

In this section, we analyze the quality of the categories (semantic classes) assigned to main entities through classification (presented in Section 6.8). We inspected few main entity instances and ranked the assigned category' relevance **high (H)**, **medium (M)** or **low (L)**. We graded H if the category describes the entity well, e.g., a `Person` or `Author` category for authors, `CreativeWork` or `Publication` for articles, etc. We graded M acceptable (if sub-optimal) categories, and L any clear misclassification, e.g., `Place` instead of `Person`, as well as `Thing` (not enough insight to classify). Tables 6.8, 6.9, 6.10 and 6.11 show for each dataset and main entity, the assigned category, and the relevance value r. As in Tables 6.4, 6.5, 6.6 and 6.7, names in *italics* are those we manually chose for collections that lack a kind name.

Dataset name	$\|C\|$	$\|\mathcal{ME}\|$	$\|\mathcal{MR}\|$	*cov*	$\mathcal{ME}$	d_{max}	$\|\mathcal{ME}_i\|$
Mondial ⟳	168	5	8	0.85	City	3	3,152
					Province	3	1,455
					Country	4	231
					Organization	4	168
					River	4	135
PubMed	26	1	0	1.0	PubMedArticle	5	957
XMark1 ⟳	136	5	10	0.91	Person	4	25,500
					Item	7	21,750
					Open_Auction	8	12,000
					Closed_Auction	8	9,750
					Category	2	1,000
XMark4 ⟳	136	5	10	0.90	Person	4	102,000
					Item	7	87,000
					Open_Auction	8	48,000
					Closed_Auction	8	39,000
					Category	2	4,000
Wikimedia	59	2	0	1.0	Page	4	54,750
					Namespace	3	32

Table 6.4: Main entities found in the XML application datasets.

Dataset name	$\|C\|$	$\|\mathcal{ME}\|$	$\|\mathcal{MR}\|$	*cov*	$\mathcal{ME}$	d_{max}	$\|\mathcal{ME}_i\|$
LDBCsmall ⟳	61	4	9	1.0	Post	6	3,189
					Comment	6	471
					Forum	9	381
					Person	6	50
LDBC0.3 ⟳	82	6	16	1.0	Comment	9	523,222
					Post	6	324,825
					Forum	16	31,097
					Tag	4	16,080
					Organization	6	7,955
					Person	13	3,514
Movies250K ⟳	38	4	3	1.0	playedIn	9	1,758,142
					Actor	12	1,099,489
					Movie	8	250,000
					Director	10	124,818

Table 6.5: Main entities found in the PG application datasets.

Dataset name	$\|C\|$	$\|\mathcal{ME}\|$	$\|\mathcal{MR}\|$	*cov*	$\mathcal{ME}$	d_{max}	$\|\mathcal{ME}_i\|$
BSBM4M ↻	340	6	7	1.0	Offer	3	226,800
					Review	5	113,400
					Product	5	11,340
					ProductFeature	3	10,519
					Producer	3	232
BSBM16M ↻	724	6	7	1.0	Review	3	1,324,146
					Offer	3	914,000
					Person	3	134,570
					Product	5	45,700
					Producer	3	921
					Vendor	3	464
Conferences ↻	29	2	2	0.83	Author	5	20
					Paper	3	10
EnelShops	46	1	0	1.0	*Shop*	6	1,136
Foodista ↻	49	4	20	0.47	Recipe	5	32,782
					Food	3	7,651
					Tool	3	150
					PreparationMethod	3	149
LUBM1M	108	5	3	1.0	Publication	11	60,342
					UndergraduateStudent	5	59,437
					GraduateStudent	6	9,259
					ResearchAssistant	6	5,454
					TeachingAssistant	6	4,156
Nasa ↻	61	5	13	0.95	Spacecraft	5	6,692
					Launch	5	5,090
					Image	3	303
					MissionRole	3	142
					Person	3	59

Table 6.6: Main entities found in the RDF application datasets.

Dataset name	$\|C\|$	$\|\mathcal{ME}\|$	$\|\mathcal{MR}\|$	*cov*	$\mathcal{ME}$	d_{max}	$\|\mathcal{ME}_i\|$
CoreResearch ↻	48	1	0	1.0	*Notice*	5	39,767
GitHub ↻	343	1	0	1.0	*Repository*	8	30
NYTimes	116	1	0	1.0	*Document*	9	4,482
Prescriptions	4,814	1	0	1.0	*Prescription*	3	239,930
Researchers ↻	25	1	1	1.0	*Researcher*	5	38,090
YelpBusiness	122	1	0	1.0	*Business*	4	150,346
YelpCheckIn	6	1	0	1.0	*Check-in*	3	131,930
YelpTips	12	1	0	1.0	*Tip*	3	908,915

Table 6.7: Main entities found in the JSON application datasets.

Dataset	$\mathcal{ME}_i$	category	r
Mondial	City	City	**H**
	Province	Province	**H**
	Country	Country	**H**
	Organization	Organization	**H**
	River	River	**H**
PubMed	PubMedArticle	CreativeWork	**H**
XMark1	Person	Person	**H**
	Item	Product	**H**
	Open Auction	Product	M
	Closed Auction	Product	M
	Category	Thing	L
XMark4	Person	Person	**H**
	Item	Product	**H**
	Open Auction	Product	M
	Closed Auction	Product	M
	Category	Thing	L
Wikimedia	Page	Thing	L
	Namespace	Thing	L

Table 6.8: Quality of $\mathcal{ME}$ classification for XML application datasets.

Dataset	$\mathcal{ME}_i$	category	r
LDBCsmall	Post	Thing	L
	Comment	Comment	**H**
	Forum	Blog	**H**
	Person	Person	**H**
LDBC0.3	Comment	Comment	**H**
	Post	Thing	L
	Forum	Blog	**H**
	Tag	Thing	L
	Organisation	Organisation	**H**
	Person	Person	**H**
Movies250K	playedIn	Play	M
	Actor	Actor	**H**
	Movie	Movie	**H**
	Director	Person	**H**

Table 6.9: Quality of $\mathcal{ME}$ classification for PG application datasets.

Dataset	$\mathcal{ME}_i$	category	r
BSBM4M	Offer	Offer	**H**
	Review	Review	**H**
	Product	Product	**H**
	ProductFeature	Product	**H**
	Producer	Thing	L
BSBM16M	Review	Review	**H**
	Offer	Offer	**H**
	Person	Person	**H**
	Product	Product	**H**
	Producer	Thing	L
	Vendor	Thing	L
Conferences	Author	Person	**H**
	Paper	CreativeWork	**H**
EnelShops	*Shop*	Restaurant	M
Foodista	Recipe	Recipe	**H**
	Food	Food	**H**
	Tool	Thing	L
	PreparationMethod	Thing	L
LUBM1M	Publication	CreativeWork	**H**
	UndergradStudent	Person	**H**
	GradStudent	Person	**H**
	ResearchAssistant	Person	**H**
	TeachingAssistant	Person	**H**
Nasa	Spacecraft	Spacecraft	**H**
	Launch	LaunchPad	**H**
	Image	ImageObject	**H**
	MissionRole	SpaceMission	**H**
	Person	Person	**H**

Table 6.10: Quality of $\mathcal{ME}$ classification for RDF application datasets.

Dataset	$\mathcal{ME}_i$	category	r
CoreResearch	*Notice*	CreativeWork	**H**
GitHub	*Repository*	Thing	L
NYTimes	*Document*	CreativeWork	**H**
Prescriptions	*Prescription*	CreativeWork	M
Researchers	*Researcher*	Person	**H**
YelpBusiness	*Business*	Thing	L
YelpCheckIn	*Check-in*	Thing	L
YelpTips	*Tip*	Thing	L

Table 6.11: Quality of $\mathcal{ME}$ classification for JSON application datasets.

Out of 68 main entities, 76.4% (52) obtained an informative category through classification: 86.5% (45) are rated H, and 13.5% (7) are rated M. In contrast, 23.6% (16) collections were classified as Thing, among which: 12 have informative kind names, and 4 (the JSON ones) do not. In all L-rated results, the linguistic and semantic signal from entities is below the algorithm's thresholds (recall Algorithm 12). This happens for two reasons: (*i*) an entity has very few properties with not-so-informative names, e.g., Yelp check-ins have only `user_id` and `date`; and/or (*ii*) most (or all) of the entity's properties have no equivalent in $\mathcal{P}$, e.g., GitHub properties such as `allow_forking` and `ssh_url`.

A classification example illustrates the combined effect of property matching and collection-label entity profiles (respectively, step (3) and (4) in Section 6.8). The *Notice* entity has 20 properties, such as `publisher`, `downloadUrl` and `doi`. Values of the `publisher` property include the Organization and Person entity types. Classification identifies three $\mathcal{P}$ properties matching the data property `publisher`: `editor`, `illustrator` and `publisher`, which vote toward the following classes: `Creative Work`, `Written Work`, `Work` and `Book`. However, `publisher` adds more support for `Creative Work`, since the collection-label entity profile for *Notice* and `publisher` (Section 5.3.1) matches the `publisher` data property range, raising the $score(C_i, dp_i, p)$ value in favor of Creative work, which is good in this case. In other cases, the classification is due to $sim(dp_i, p)$ and $\mathcal{P}$ domain typing, e.g., for `PubMedArticle`, the properties `PubMedLink`, `DOI` and `KeywordList` all voted toward Creative work; for XMark items, `seller` voted toward the `Offer`, `Order`, `BuyAction`, `Demand`, `Flight` and `Product` categories, however, `quantity` only voted for `Product` and helped it be selected.

In conclusion, **classification was overall successful**, leveraging the property names, collection-label entity profiles, and the background semantic information ($\mathcal{K}$ and $\mathcal{P}$). In general, classification quality depends on the overlap between the dataset vocabulary, and the background information: categories may not be found for very specific entities, e.g., GitHub. Even when the classification is not very precise, e.g., Creative work for (medical) Prescriptions, it is still useful (in particular in this example, where the collection has no kind name to guide users).

6.9.5 Scalability of the abstraction computation

We now study the performance of data abstraction as a function of the input data size.

We measure the execution time of all the steps involved in the abstraction computation method using (w_{dw-PR}, $bound_{fl-ac}$). To study this, we use four of our synthetic, controlled-size datasets: XMark (XML), Researchers (JSON), BSBM (RDF), and Movies (PG), and show the results in Figure 6.13; all axes are logarithmic. The normalization step, i.e., transforming the labeled graph G_0 into an unlabeled graph G, is longer for RDF than other data models; this is because all RDF data edges are labeled compared to others. Identifying the main entities takes most time; this is due to the cost of updating the graph (excluding nodes as described in Section 6.5.2) before recomputing weights. Main entity identification is faster on the JSON dataset as there are few entities selected, thus few graph updates. The time taken to identify relationships (Section 6.6) is negligible (thus not shown). This is because this task only manipulates the collection graph, much smaller than the graph.

All our abstraction methods scale up linearly in the data size. The methods using $\text{bound}_{\text{desc}}$ and $\text{bound}_{\text{leaf}}$ (not plotted to avoid clutter) are faster, since they do not recompute weights. However, as shown in Section 6.9.2, their results are of lower quality.

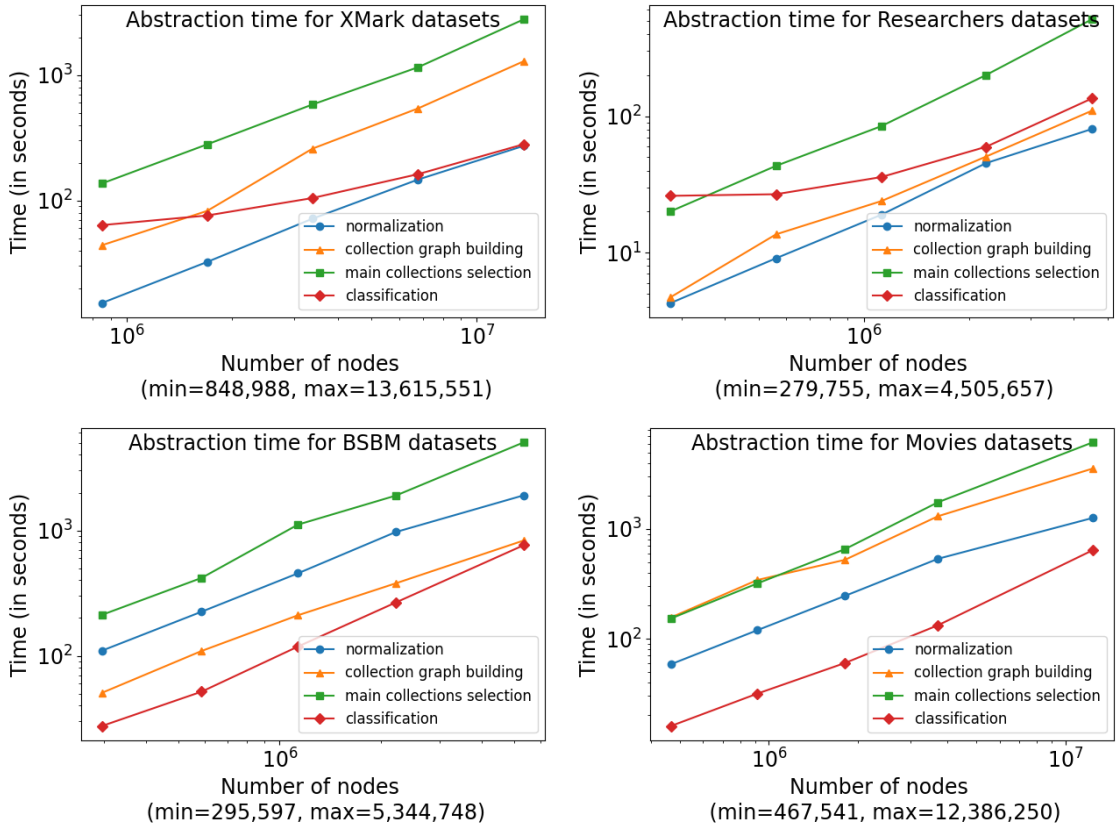


Figure 6.13: Abstraction computation times on synthetic XML, JSON, RDF and PG datasets, using (w_{dw-PR}, $bound_{fl-ac}$).

Dataset name	Schema extractor	# nodes	# edges
CoreResearch	[19]	128	-
	[157]	146	
Prescriptions	[19]	7,221	-
	[157]	4,218	
YelpTips	[19]	191	-
	[157]	144	
YelpBusiness	[19]	18	-
	[157]	21	
YelpCheckIn	[19]	12	-
	[157]	12	
LUBM	[143]	23	187
LDBC	[34]	10	22

Table 6.12: Schemas sizes for a subset of JSON, RDF and PG evaluation datasets.

6.9.6 Inferred schemas vs. abstractions

Abstractions have different goals and uses than dataset schemas. Schemas define a notion of data *validity*, and are *used by code* that processes the data; abstractions aim to give *human users a simple, first glance* at the data. Schemas have *technical features*, such as inheritance and property cardinalities, that abstractions voluntarily leave out. They contain interconnected types, while abstractions *wrap some collections into the main entity boundaries*, to facilitate understanding. At the same time, abstractions share with schemas, in particular those *inferred from the data*, e.g., [19, 20, 157, 33, 34, 143], the goal to compactly describe a dataset.

Keeping this in mind, we tested recent schema extractors for: JSON [19, 157], RDF graphs [143], and PGs [33, 34]. We report in Table 6.12 results on some datasets from Table 6.1. For **JSON**, on the CoreResearch, Prescriptions, YelpTips, YelpBusiness and YelpCheckIn datasets, respectively, [19] and [157] produce schemas of dozens to several thousand nodes. The schema itself is a JSON document, and we count its nodes as a measure of its complexity. Clearly, schemas of more than hundreds of nodes are unsuited as abstractions. For **RDF**, the LUBM schema [143] features 23 node shapes (types), one for each RDF type, and 187 property shapes (an RDF property such as `worksFor` leads to 9 property shapes: `worksForProfessor`, `worksForChair`, etc.). Such precision leads to many syntactic details, going against our need for clarity. For **PGs**, on the LDBC dataset, [34] creates a schema of 10 types connected by 22 edges. While compact, it contains several nodes for the same concept, e.g., three nodes labeled "Post", but with slightly different properties. Our abstractions are more compact (Tables 6.4, 6.5, 6.6 and 6.7), and prefer node kinds over structural precision.

These results confirm that **compact abstractions are needed to give human users a first idea of a dataset**. They helpfully **complement schemas**, whose objectives are different.

6.9.7 Remarks on abstraction

During our experiments, presented in Section 6.9, we have observed some interesting particular cases, which we discuss below.

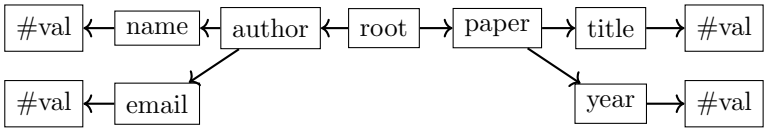


Figure 6.14: A collection graph leading to a disconnected Entity-Relationship schema.

Boundary overlap

Boundaries of several collections may overlap, e.g., `conf` and `author` collections have in their boundaries the collection `name`. As discussed in Section 5.3.4, the collection graph introduces approximations, e.g., the `name` nodes were used in two different contexts, which are conferences and authors. However, all `name` nodes are represented in the same collection in $\mathcal{G}$. This approximation also appears in the boundaries: both the `conf` and `author` main entities do have `name` in their boundaries.

Result variations

Data abstraction results may differ depending on the scoring and boundary method used, as well as the parameters E_{max}, cov_{min} and f_{min}. Relational Entity-Relationship (E-R) modeling is also known to include a subjective factor, and for a given database, several E-R models may be correct. Our focus is on not missing any essential component of the dataset, while allowing users to limit the amount of information through E_{max}, and classifying the main entities into semantic categories.

E-R schemas connectivity

Some abstractions may produce disconnected Entity-Relationship schemas, even if the initial data was connected, e.g., an XML or a JSON document. For instance, Figure 6.14 shows the collection graph obtained from an XML document describing authors and papers. The main entity selection algorithm will report the collections of authors and papers: the final Entity-Relationship schema will contain two entities with no relationship.

When designing an Entity-Relationship diagram for an application database, it is very rare to find unconnected entities. This is also because the general practice is to *factorize as much as possible* in order to obtain *atomic* entities.

JSON abstractions

For the particular case of data abstractions computed out of JSON documents, it appears that only *maps* can be elected as main entities. Indeed, only collections with an internal structure, i.e., data properties and values are eligible (recall eligibility criteria in Section 6.2). On the contrary, arrays, map keys and (map or array) values cannot be reported.

6.9.8 Experiment conclusion

The abstraction method (w_{dw-PR}, $bound_{fl-ac}$) attains the best results overall, even on complex, cyclic collection graphs. This method successfully identifies the central dataset entities, when their structure ranges from simple (depth 2 in Table 6.4) to very complex (depth 16 in Table 6.5). Classification, with record kind names, property names, collection-label entity types, and semantic

resources, is overall successful in identifying what each main entity is about. ABSTRA sets parameters' default values to w_{dw-PR}, $bound_{fl-ac}$ and thresholds in Section 6.9.1. Our abstractions are computed in linear time in the input size. Intuitive, and more compact than (inferred) schemas due to their focus on structured entities (as opposed to node types), they provide useful first-glance dataset summaries.

6.10 Summary

In this chapter, we have presented the last important stage toward computing, from a dataset, a user-friendly diagram, akin to Entity-Relationship ones, which we call dataset abstraction. Two core technical challenges needed to be addressed in order to achieve this. The first is to interpret the collection graph (Chapter 5), separating collections which we consider entity roots, from those that we consider to be their attributes. This task is hard for several reasons: (*i*) the potentially complex structure of the collection graph, with multiple cycles; (*ii*) collections shared among many others; (*iii*) our target entities may have a deeply nested structure.

The second technical challenge is to manage to automatically attach a semantic class to each entity, in order to give users a first understanding of what the entity contains. We achieve this by leveraging linguistic signal in the types and labels of nodes and edges encompassed in the collection structure; further, we take advantage of the Named Entities present in our data graph. Our experiments have demonstrated the feasibility and the practical interest (for users) of our abstractions.

Having finalized our presentation of dataset abstraction, we turn to another exploitation of the collection graph to help user discover heterogeneous data.

7

Entity-to-entity path exploration

Chapter Outline

Chapter Abstract. In this chapter, we study a different approach for helping users discover interesting information in semi-structured datasets. Specifically, we identify, rank, and evaluate *graph paths connecting pairs of Named Entities* with the help of the collection graph (Chapter 5). We start by explaining how entity-to-entity[1] data paths exist in data graphs, like the one shown in Figure 5.8 (Section 7.1). Next, we discuss the importance of highly reliable NE extractors in our context, and how we devised a novel, more accurate one based on OpenAI's ChatGPT model (Section 7.2). Further, we describe how we rank entity-to-entity data paths, based on metrics we propose as part of this thesis. We also describe how, from few user interactions, we enumerate (only) data paths that are of interest to them (Section 7.3). To speed up data paths evaluation on the underlying data graph, we recommend a set of views (sub-paths) to materialize, and rewrite each data path using these materialized views (Section 7.4). Finally, we present an experimental evaluation of these methods (Section 7.5).

[1]Here, the concept of entity refers to Named Entities, as described in Section 4.2; do not confuse them with Abstra entities.

7.1 From data graphs to entity-to-entity data paths

Building on entity-rich data graphs, users may be interested in finding connections between Named Entities of interest, as in "are Helmut Greim and Monsanto related in some way?", "how do French politicians relate to the top-40 most influential French companies?", "how people are connected to companies?", etc. Toward this, our work leverages the idea that we can find entity-to-entity data paths, or simply data paths, starting and ending in Named Entities of interest to the user. Because it is often hard to specify the Named Entities one can expect in a dataset, we instead prefer to ask users the Named Entity types (recall $\mathcal{T}$, which include Person, Location, Organization, etc) that are of interest to them. Among the (numerous) data paths, some are not of high quality. Therefore, **our goal is to enumerate and rank a set of interesting entity-to-entity data paths connecting two Named Entity nodes at the path extremities**. To chase this goal, we establish the following set of requirements:

(R1) The entity-to-entity path exploration system should work on any data graph and regardless of the edge directionality;

(R2) Most reliable and interesting entity-to-entity paths should be shown first;

(R3) Users should be able to specify their interest in an interactive way.

7.2 ChatGPT-based Named Entity extractor

The main ingredient to enumerate reliable data paths (recall requirement (R2)) is to have top-quality NER modules, identifying all expected NEs (not more, not less) and assigning them the right type. In reality, NE extractors may fail in identifying some entities, because it may be rare in the training data they have seen; or the data may contain tokens that are miss-understood (such as proper nouns without capitalization, etc). As shown in ConnectionLens [13], the Flair extractor has significantly better accuracy than the Stanford one. However, a close examination of extraction results shows many to be *false positives* (NEs extracted when there should be none). This likely is because the news training corpus differs from the text snippets found in various semi-structured datasets we experimented with. While some false positives (and/or negatives) are to be expected in NLP settings, false positives lead to erroneous extraction edges. In our work, in turn, these edges lead to erroneous entity-to-entity data paths. To solve the problem of false positives with extractors based on pre-trained language models, one could retrain the Flair model for different corpora, but this is labor-intensive. Instead, we have developed **a new NE extractor, leveraging OpenAI's ChatGPT 4**[2]. Its very large model, trained on a huge corpus, allowed us to expect good performance, once properly prompted, in detecting NEs in texts of various forms even without corpus-specific fine-tuning.

We used ChatGPT in a question-answer mode. Each question we send, also called *prompt*, consists of a fixed directive, together with a string ($\mathscr{L}$-leaf in the dataset) in which we ask ChatGPT to identify Named Entities. For each entity found, ChatGPT also returns a type it considers the most suited for the entity in the context of the input string, as well as its confidence. There is no way of knowing the complete set of entity types that ChatGPT could propose; in ConnectionLens, on the other hand, we use a fixed set of entity types $\mathcal{T}$, and extraction based on trained models is

[2] https://platform.openai.com

> Please get each named entity you identify in the following string: 'XXX'. Return a table containing four columns, one for the named entity name, one for the named entity type you assigned to it, one for a category among (person, organization, location) that fits your type, and one with the confidence you have in the category assigned to the extracted entity where the confidence is a float value between 0 and 1. If no category fits your type or a sub-type of your type, set the category as OTHER. If no named entities of the expected types, answer NONE.

Figure 7.1: ChatGPT prompt for NE extraction (directive plus a string, denoted 'XXX').

Named Entity	Type	Category	Confidence
Y. Hu	Person	Person	0.95
A. Coates	Person	Person	0.90
antibiotic resistance breaker technology	Product	OTHER	0.70
quinoline and tobramycin	Product	OTHER	0.70
Pseudomonas spp.	Species	OTHER	0.80
Helperby Therapeutics	Company	Organization	0.85
HT61	Product	OTHER	0.60
Dr Richard Amison	Person	Person	0.80

Table 7.1: Sample ChatGPT NE extraction results.

only used to identify People, Organizations, or Locations (recall Section 4.2). Thus, our directive also instructs ChatGPT to map its entity types to our desired types, or to an extra category OTHER. The best prompt we found appears in Figure 7.1[3]. It constrains the answer format to ensure a deterministic structure that facilitates its integration. For instance, Table 7.1 shows the NEs (with their attached information) found by ChatGPT in the string: *"Declaration of competing interest Y. Hu and A. Coates are the coinventors of the antibiotic resistance breaker technology, in particular the combination of the quinoline and tobramycin (patent granted). They were the first to test this combination against highly resistant Pseudomonas spp. They originated the concept and performed the background work upon which this work is based. A. Coates, Y. Hu and CP declare they have equity in Helperby Therapeutics who are developing HT61. CP is in receipt of a grant from Helperby Therapeutics to support Dr Richard Amison for the conduct of the in vivo aspect of this study. There are no other conflicts of interest to declare"*. This string is part of the metadata (Conflict of Interest statement) in a PubMed article; we analyzed such statements in a previous study [12], using ConnectionLens and the Flair NE extractor. As Table 7.1 shows, our prompt is effective in getting ChatGPT to perform high-quality extraction. We analyze its performance in more depth in Section 7.5.2.

[3]We have tested 6 other prompts, which were not giving sufficiently precise instructions and/or structured answer format, before reaching the one of Figure 7.1.

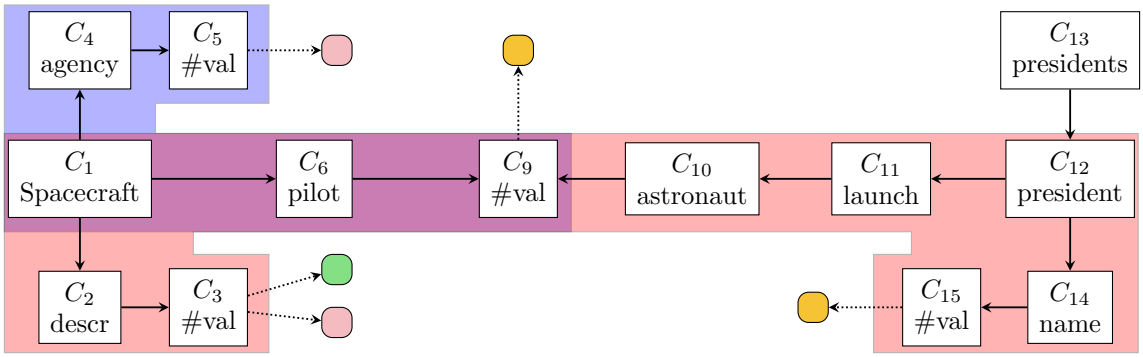


Figure 7.2: Multi-dataset collection graph corresponding to Figure 4.5. Highlighted areas correspond to entity-to-entity paths and their intersection (in purple) will be materialized as a view.

7.3 Entity-to-entity path enumeration and associated path metrics

Based on a data graph with highly accurate Named Entities, we are now interested in finding interesting paths into it. However, enumerating them on the data graph itself would be prohibitively expensive, we instead rely on the collection graph (recall Chapter 5) for this task. In the set of enumerated paths, some of them may be more interesting than others. This is why we *rank* them, based on their extraction reliability ("*how confident are we in the Named Entities?*") and their force ("*is the information diluted?*"). Last but not least, we efficiently evaluate the set of paths that are of interest to the user, leveraging a multi-query optimization approach.

Figure 7.2 illustrates the collection graph obtained from the data graph shown in Figure 5.8 and represents Named Entity types extracted from $\mathscr{L}$-leaf collections as colored squares, e.g., ■ represent people extracted from C_9 and C_{15}. The colored areas correspond to paths connecting People to Organization entities. The purple sub-path will be later materialized as a view (Section 7.4) to speed up their evaluation.

7.3.1 Entity-to-entity collection path enumeration

Following requirement (R3), we first ask users few questions to guide/restrict the path enumeration toward paths that are of interest to them:

- A **pair of Named Entity types** (τ_1, τ_2), where $\tau_1, \tau_2 \in \mathcal{T}$, the set of supported NE types (recall Section 4.2).
- Possibly, the **maximum path length** $L_{\max}$, i.e., the number of edges it contains, whose default value we set to 10. Depending on the application, interesting connections can be made by paths of different lengths; however, it appears likely that beyond a certain length, connections may become meaningless.

Next, and following requirement (R1), we seek to efficiently enumerate the set of data paths in G connecting two NEs, respectively of types τ_1 and τ_2, and of length at most $L_{\max}$. However,

enumerating data paths on the data graph G itself would be prohibitively expensive and slow, and even unfeasible for large data graphs. Therefore, we will work with $\mathcal{G}_U$, the collection graph whose edges are undirected. Recall from Section 5.3.3: a collection path is of the form $cp = (C_i \rightarrow C_a, C_a \rightarrow C_b, ..., C_b \rightarrow C_j)$, which we can simply abbreviate in $cp = (C_i \nleftrightarrow C_j)$. We extend collection paths to entity-to-entity collection paths as follows:

Definition 7.3.1 (*Entity-to-entity collection path*)

Given a collection graph $\mathcal{G}$, an **entity-to-entity collection path** $ecp \in \mathcal{G}$ is a sequence of collection edges such that $ecp = \square \dashleftarrow C_i \nleftrightarrow C_j \dashrightarrow \blacksquare$ where $\square, \blacksquare$ are two collection-label entity profiles and C_i, C_j are value collections. The directions of the leftmost and rightmost edges are by convention always toward $\square$ and $\blacksquare$, which represent entities.

For instance, an entity-to-entity collection path in Figure 7.2 would be $\blacksquare \dashleftarrow C_3 \leftarrow C_2 \leftarrow C_1 \rightarrow C_6 \rightarrow C_9 \dashrightarrow \blacksquare$.

First, we build $\mathcal{G}_U$, the undirected collection graph, and (efficiently) enumerate its set $\mathcal{P}$ of collection paths using Algorithm 1. Note that, because $\mathcal{G}_U$ is undirected, enumerated paths may contain collection edges in both directions, i.e., $\rightarrow$ and $\leftarrow$. Next, to build $\mathcal{EP}$, the set of entity-to-entity collection paths, we take from $\mathcal{P}$ only those meeting the user requirements: $\tau_1 \in \square$, $\tau_2 \in \blacksquare$ and its length is lower than $L_{\max}$.

Next, we compute the directionality (Section 7.3.2), reliability (Section 7.3.3) and force (Section 7.3.4) of each path thus enumerated, and rank them, first by reliability (truncated to 2 decimals) and then by force. Then, we inform users about paths and they can then chose a set of entity-to-entity collection paths to materialize, e.g., "only the shared-root ones", or "the ones ranked in the top-20", or "just those involving specific internal node labels", etc. How paths are evaluated on the data graph will be discussed in Section 7.4.

7.3.2 Path directionality

We can classify entity-to-entity collection paths according to the directions of the edges in $\nleftrightarrow$. Specifically, paths may be:

- *Unidirectional*, i.e., all *ecp* edges go from C_i toward C_j, or the opposite;
- *Shared-sink*, i.e., *ecp* may contain a (collection) node C_k such that all edges between C_i and C_k (if any) go from C_i toward C_k, and all edges between C_j and C_k (if any) go from C_j toward C_k. A shared-sink path is $C_1 \rightarrow C_6 \rightarrow C_9 \leftarrow C_{10} \leftarrow C_{11} \leftarrow C_{12}$.
- *Shared-root*, i.e., *ecp* may contain a (collection) node C_k such that all edges between C_k and C_i (if any) go from C_k toward C_i, and all edges between C_k and C_j (if any) go from C_k toward C_j. A shared-root path is $C_3 \leftarrow C_2 \leftarrow C_1 \rightarrow C_6 \rightarrow C_9$.
- *General*, i.e., the edges may be in any direction.

Unidirectional paths are quite rare. This is because entity-connecting paths must have at each end a node from which an entity is extracted. Most of the time, these are *two literal (string) nodes* (as opposed to internal nodes structuring the dataset). Literals have incoming edges, but not outgoing ones (other than those toward extracted entities); thus, *there is no unidirectional path from a literal to another*. However, in some RDF datasets, *NEs are extracted from URIs*, e.g., the triple

⟨`dbpedia:Facebook`⟩⟨`dbpedia:locatedIn`⟩⟨`dbpedia:California`⟩ is a unidirectional data path from an Organization to a Location. Similarly, shared-sink paths only occur when nodes in C_i and C_j have outgoing edges, and NEs appear in their labels; this only happens in RDF URIs.

7.3.3 Path reliability

Data graph edges can be divided into: (*i*) structural, which originate in the way the data is organized in the input graph; in our approach, we consider such edges as certain, or fully reliable; and (*ii*) extraction edges, which connect a NE to the string from which it had been extracted. In turn, an extraction edge can reflect a *true positive* (the entity is correctly extracted, i.e., most human users would agree that the entity of the respective type is present in the string), or a *false positive* (a human user would not consider the entity is present there). In the collection graph, we call *extraction edge* an edge corresponding to one or more data graph extraction edges. Each entity path has at least two extraction edges, $\square \leftarrow C_i$ and $C_j \rightarrow \blacksquare$; if the path is either shared-sink or general, it may contain other extraction edges. In Figure 7.2, the path $\blacksquare \dashleftarrow C_3 \leftarrow C_2 \leftarrow C_1 \rightarrow C_6 \rightarrow C_9 \dashrightarrow \blacksquare \dashleftarrow C_{10} \leftarrow C_{11} \leftarrow C_{12} \rightarrow C_{14} \rightarrow C_{15} \dashrightarrow \blacksquare$ in the collection graph exhibits four extraction edges.

Examining several datasets, we noted situations when *all* the data extraction edges behind a given collection graph extraction edge e correspond to false positives, due to NER errors. For instance, in PubMed bibliographic data, chemical acronyms in article titles were mistakenly extracted as being Organizations. In more subtle cases, people names were extracted from (*i*) titles; two among a few thousand articles were scientists' obituaries thus had their name in the title; (*ii*) affiliations, when a research lab, institution, or a street is named after a person. In such cases, a person name is technically present, but since most titles, and most affiliations, do not feature people, we consider that the collection-level extraction edge is not *reliable*.

Formally, an extraction collection edge has a **reliability**, denoted e_{rel}, as follows:

Definition 7.3.2 (*Extraction collection edge reliability*)

Given an extraction collection edge $e \in \mathcal{G}$ of the form $C_i \dashrightarrow \blacksquare$ where $\blacksquare$ corresponds to NEs of a specific entity type τ, its **reliability** is:

$$e_{\text{rel}} = \frac{|\{n \in C_i | n \dashrightarrow \tau\}|}{|C_i|}$$

Further, we compute the **reliability of an entity-to-entity collection path** as follows:

Definition 7.3.3 (*Entity-to-entity path reliability*)

Given $ecp \in \mathcal{G}_U$, an entity-to-entity collection path, its **reliability** is:

$$ecp_{\text{rel}} = min(\{e_{\text{rel}} | e \in ecp\})$$

Using the minimum to aggregate extraction edge reliability is a conservative choice, which penalizes a path according to its least reliable edge. In our experiments, this choice gave good results; indeed, even a single unreliable edge in a collection graph path may make it meaningless.

7.3.4 Path force

Beyond directionality and reliability, entity paths can also be analyzed based on the numbers of edges adjacent to graph nodes. ConnectionLens [13] attaches to each data edge $e \in G_0$, having a non-empty label a, a measure called *specificity*. Let e be the edge $n_1 \xrightarrow{a} n_2$ for some nodes n_1, n_2. The *specificity* of e, denoted e_s, is computed as $\frac{2}{(N_{1,a}+N_{2,a})}$, where $N_{1,a}$, $N_{2,a}$ are the numbers of edges labeled a outgoing n_1, respectively incoming n_2. The highest $N_{1,a}$ and/or $N_{2,a}$, the lowest e_s. For instance, in the data graph G_0 used to compute the normalized data graph G in Figure 5.8, the specificity of the two `agency` RDF labeled edges would be $\frac{2}{(1+2)} = \frac{2}{3}$. For our purposes, we extend specificity to unlabeled edges as follows: the specificity of an edge $n_1 \xrightarrow{\epsilon} n_2$ is $\frac{2}{(1+n_{1,2})}$ where $n_{1,2}$ is the number of ϵ (empty-labeled) edges outgoing n_1, toward nodes having the same label as n_2. For instance, the specificity of the edge $N_6 \rightarrow N_7$ is also $\frac{2}{3}$ in Figure 5.8. In [13], edge specificity has been used as an ingredient for scoring connections (paths or trees) in the original graph, returned when users search the graph using keywords. Indeed, low-specificity edges can be seen as "weakening" connections, e.g., when a person has 1 spouse and 200 friends, intuitively, an edge (thus, path) going from the person to a friend is weaker than one going from the person to the spouse.

In [22], we leveraged data edge specificity as follows:

1. In the collection graph, the edges with a non-empty label, connecting nodes from two equivalence classes, lead to a collection, e.g., `agency` triples lead to C_4. To this *collection* is attached the **average specificity** of all the *data edges it comes from*, e.g., to C_4 corresponds $\frac{2}{3}$. Empty-label edges connecting graph nodes from two equivalence classes lead to an edge in the collection graph, e.g., $C_{11} \rightarrow C_{10}$. To an edge between collections is attached the **average specificity** of the *original edges*.

2. *Paths with low-specificity collections or edges were pruned.* Specifically, users were first asked to state how many low-specificity collections and collection edges they are willing to review, then shown the lowest-specificity ones, each of which they can validate or invalidate. Then, we only enumerated paths that did not traverse invalidated collections. This approach allowed to control the effort required from users; it is also more accurate than just invalidating collections whose specificity is below a certain threshold, which could lead to decisions suitable for some collections but unsuitable for others. However, this approach of [22] still had drawbacks. On one hand, it required user effort; on the other hand, low-specificity collections and collection edges, that do not make it to the users' inspection may be preserved, leading to weak paths.

Toward avoiding these limitations, we change our approach, as follows. First, we *no longer* require users to inspect structural metrics. Second, we use structural metrics to *rank paths, instead of pruning them*. Third, we *compute a different structural metric*, the path force. We start by defining a G data edge cardinality as:

Definition 7.3.4 (*Data edge cardinality*)

Given a data edge $e = N_i \rightarrow N_j$ such that $n_i \in C_i, n_j \in C_j$, its **edge cardinality** e_{card} is:

$$e_{\text{card}} = |\{N_i \rightarrow N_z | N_z \in C_j\}|$$

This differs from specificity in several aspects. First, cardinality is asymmetric, i.e., it only considers how many edges exit a node in C_i, not how many enter a particular node in C_j. This seems simpler and more intuitive. Second, cardinality interprets the neighborhood of n_i through the prism of the collections C_i, C_j to which n_i, n_j belong. Specificity had no awareness of collections, and only focused on edges exiting n_i and entering n_j.

Next, we define the **collection edge force** $f(C_i \to C_j)$ of a collection edge as the inverse of the maximum cardinality among all data edges represented by the collection edge; this number is in $(0, 1]$. Taking the inverse of the maximum cardinality penalizes the existence of even one node $n_i \in C_i$ having a large number of edges to nodes from C_j. Averaging specificity as in [22] allowed to smooth the impact of such nodes. We prefer the inverse of the maximum cardinality since we consider a C_i node with many edges to C_j signals that connections $C_i \to C_j$ may be not too selective for a C_i node, i.e., one *could* have hundreds of friends, or written hundreds of papers, even if most people do not. In contrast, one always has a single birth country, at most one or a few spouses over a lifetime, etc. Finally, we define the **path force** $F(p)$ of a path p as:

Definition 7.3.5

Given an entity-to-entity collection path $ecp \in \mathcal{G}_U$, its **force** $F(ecp)$ is:

$$F(ecp) = \prod_{C_i \to C_j \in ecp} f(C_i \to C_j)$$

This combines all forces along the path, penalizing multiple and/or low values. It also penalizes paths containing many edges whose force is below 1.

7.4 Data paths materialization

At this point, we have a set of *entity-to-entity collection paths*, which must be transformed into queries and evaluated *on the data graph*. Each such query matches similar-structure data paths, thus its results are shown to users as a table: the first and last attribute of such a table comprise entities of type τ_1, τ_2, while the intermediary attributes are the nodes and edges connecting these entities in the data graph. For instance, let τ_1 be Person, τ_2 be Organization: the light-blue and light-red background shapes in Figure 5.9 materialize two paths which, in this graph, connect the pink child of C_5 (■) with the yellow children (■) of C_9, respectively, those of C_3 and C_{15}.

7.4.1 From a collection path to a query over the data graph

Each entity-to-entity collection path translates into a chain-shaped conjunctive query. For instance, the path on blue background in Figure 5.9, going through C_5 and C_9, becomes:

$$q_1(\bar{x}) \text{ :- } n(x_1, \tau_{\text{Org}}, \blacksquare), e(x_2, x_1, \_), n(x_2, \_, C_5), e(x_3, x_2, \texttt{agency}), n(x_3, \_, C_1), e(x_3, x_4, \texttt{pilot}), n(x_4, \_, C_9), e(x_4, x_5, \_), n(x_5, \tau_{\text{Person}}, \blacksquare)$$

This query refers to two relations: $n(ID, type, coll)$, describing nodes, with the last attribute denoting their collection, and $e(s, t, label)$, describing edges between nodes s and t and carrying a certain label. Each x_i is a variable; $\bar{x}$ in the query head denotes all the x_i variables, $1 \leq i \leq 5$. We use $\_$ to denote a variable which only appears once, in a single query body atom. Finally, τ_{Org} and τ_{Person}

denote the node types of extracted Organization and Person entities. Similarly, the red-background collection path translates into:

$$q_2(\bar{x}) \text{ :- } n(x_1, \tau_{\text{Org}}, \blacksquare), e(x_2, x_1, \_), n(x_2, \_, C_3), e(x_3, x_2, \texttt{descr}), n(x_3, \_, C_1), e(x_3, x_4, \texttt{pilot}), n(x_4, \_, C_9), e(x_5, x_4, \_), n(x_5, \_, C_{10}), e(x_6, x_5, \_), n(x_6, \_, C_{11}), e(x_7, x_6, \_), n(x_7, \_, C_{12}), e(x_7, x_8, \_), n(x_8, \_, C_{14}), e(x_8, x_9, \_), n(x_9, \_, C_{15}), e(x_9, x_{10}, \_), n(x_{10}, \tau_{\text{Person}}, \blacksquare)$$

Note how the above path features labeled and unlabeled edges, respectively from the RDF and XML datasets. Each of these queries can be evaluated through any standard graph database. However, evaluating dozens or hundreds of path queries on large graphs can get very costly. Further, since we do not know which paths may result from the user choices, we cannot establish path indexes beforehand.

View-based optimization To address this problem, we propose an optimization, based on the observation that *queries resulting from collection paths may share some sub-paths*. For instance, the sub-query $s(x_3, x_4)$:- $n(x_3, \_, C_1), e(x_3, x_4, \texttt{pilot}), n(x_4, \_, C_9)$ is shared by $q_1(\bar{x})$ and $q_2(\bar{x})$. Therefore, we decide to (i) evaluate s and store its results; (ii) rewrite $q_1(\bar{x})$ and $q_2(\bar{x})$ by replacing these atoms in each query, by a single occurrence of the atom $s(x_3, x_4)$. The next sections formalize this for larger query sets, also showing how to handle different *alternatives* that may arise as to which shared sub-paths to materialize.

7.4.2 Candidate views enumeration

A first question we need to solve is enumerating, based on a set $\mathcal{Q}$ of path queries, the possible sub-queries that we could materialize, and based on which we could rewrite some workload queries.

Let $q \in \mathcal{Q}$ be a path query: it is an alternating sequence of node (n) and edge (e) atoms. We denote by n_q the number of edge atoms, then the number of node atoms is $n_q + 1$. We denote by $n_{\mathcal{Q}}$ the highest n_q over all $q \in \mathcal{Q}$.

Without loss of generality, our first heuristic **(H1)** is: we only consider **connected sub-paths** of q as candidate sub-queries. If q is of the form $q(\bar{x})$:- $n_1, e_1, \ldots, e_{n_q}, n_{n_q+1}$, each connected sub-path of q, denoted sq, is determined by two integers $1 \leq i \leq n_q$, $i < j \leq n_q + 1$, such that $sq(x_i, x_j)$:- $n_i, e_i, \ldots, n_j, e_j$, and x_i, x_j are the IDs of the nodes in the atoms n_i, n_j, respectively. We denote by $q|^{i,j}$ the sub-query of q determined by the positions i, j. For instance, when q_1 is the sample query in Section 7.4.1, $q_1|^{3,4}$ is the sub-query $s(x_3, x_4)$ introduced there. Considering connected (cartesian-product free) candidate views is common in the literature too (see Section 3.3).

Each query $q \in \mathcal{Q}$ has $O(n_q^2)$ connected sub-paths, that can be easily enumerated from q's syntax. A second heuristic **(H2)** we adopt is: we only consider **shared sub-paths**, that is, those sub-paths s for which there exist $q', q'' \in \mathcal{Q}$, $q' \neq q''$, and integers i', j', i'', j''' such that $s = q'|^{i',j'} = q''|^{i'',j''}$, possibly after some variable renaming. For the queries q_1, q_2 in Section 7.4.1, the sub-query $s_{3,4}$ is $q_1|^{3,4}$ and also $q_2|^{3,4}$. (H2) restricts the number of candidate views from $|\mathcal{Q}| \times n_{\mathcal{Q}}^2$ to a number that depends on the actual workload $\mathcal{Q}$, and which decreases when $\mathcal{Q}$ paths look more like each other. Another interest of (H2) is: the benefit of using a view v to rewrite one query q is likely offset by the cost of materializing v; actual performance improvements start when v is *used twice (or more)*, which is exactly the case for sub-queries shared by several $\mathcal{Q}$ queries.

Our third heuristic **(H3)** is: among the possible sub-queries shared by two queries q', q'', **consider only the longest ones**. That is, if s_1, s_2 are two shared sub-queries of q' and q'' such that

$n_{s_1} > n_{s_2}$, do not consider the sub-query s_2.

Our heuristics (H1), (H2), (H3) lead to **building the candidate view set** $\mathcal{V}$ as follows. For each pair of distinct queries (q', q'') where $q', q'' \in \mathcal{Q}$, add to $\mathcal{V}$ the longest, shared, connected sub-queries of q' and q''. The complexity of this algorithm is $O(|\mathcal{Q}|^2 \times n_{\mathcal{Q}}^2)$, while $|\mathcal{V}|$ is in $O(|\mathcal{Q}|^2)$.

Algorithm 13: Selecting views to materialize and the respective view-based rewritings

Input : $\mathcal{Q}$: queries, $\mathcal{V}$: candidate materialized views
Output: $\mathcal{M}$: materialized views, $\mathcal{R}$: rewritings for some $\mathcal{Q}$ queries

$\mathcal{M} \leftarrow \emptyset$; $\mathcal{R} \leftarrow \emptyset$
while $\mathcal{V} \neq \emptyset$ **do**
  for $v \in \mathcal{V}$ **do**
    $ben(v) \leftarrow 0$; $cost(v) \leftarrow$ cost to compute and store the view v
    for $q \in \mathcal{Q}, q$ can be rewritten using v **do**
      $(ben(v, q), r_{q,v}) \leftarrow$ the cost of evaluating q directly on the graph, minus the cost of evaluating q based on v, through the rewriting $r_{q,v}$
      $ben(v) \leftarrow ben(v) + ben(v, q)$
  $(v_{max}, b_{max}) \leftarrow$ a view v_{max} maximizing $ben(v) - cost(v)$, and its benefit
  if $b_{max} < cost(v_{max})$ **then**
    exit
  Add v_{max} to $\mathcal{M}$
  for $q \in \mathcal{Q}, q$ *can be rewritten using* v_{max} **do**
    if $ben(v_{max}, q) > 0$ **then**
      Add $r_{q,v_{max}}$ to $\mathcal{R}$
      Remove q from $\mathcal{Q}$
  Remove v_{max} from $\mathcal{V}$

7.4.3 Materialized views selection and path queries rewriting

Knowing the path queries $\mathcal{Q}$ and the candidate view set $\mathcal{V}$, we need to determine: a set $\mathcal{M} \subseteq \mathcal{V}$ of views which we actually materialize, in order to rewrite some $\mathcal{Q}$ queries. We collect the rewriting of each such queries in $\mathcal{R}$. The decision to materialize a view incurs a *cost*, since the view data must be computed and stored. We denote $cost(\cdot)$ the cost of evaluating a view (or query), and assume it can be determined without actually evaluating it. Materializing a view is more attractive if (i) rewritings using it reduce significantly query evaluation costs, and (ii) its own materialization cost is small.

In the most general case, a query could be rewritten based on any number of views, and also involving the base graph. For instance, query q_1 from Section 7.4.1 could be rewritten as: $q_1|^{1,3} \bowtie q_1|^{3,4} \bowtie q_1|^{4,6}$, where each $\bowtie$ denotes a natural join, on the variables x_3, respectively, x_4. However, enumerating all such alternatives makes the query rewriting problem NP-hard [85]. Instead, we adopt another pragmatic heuristic **(H4)**: **rewrite each query using not more than one view**. This simple choice keeps the view selection complexity under control, all the while providing good performance.

Algorithm 13 depicts our **greedy** method for finding $\mathcal{M}$ and $\mathcal{V}$. It computes the *benefit* of each view v for each query that may be rewritten using v, as well as the cost of v. In a greedy fashion, it decides to materialize the view v_{max} maximizing the *overall* benefit (for all $\mathcal{Q}$ queries), and uses it to rewrite all queries whose evaluation cost can be reduced thanks to v_{max}, via the rewriting $r_{q,v_{max}}$. These queries are then removed from $\mathcal{Q}$, the benefits of the remaining views are recomputed over the diminished $\mathcal{Q}$, and the process repeats until no profitable view to materialize can be found. Algorithm 13 makes at most $O(|\mathcal{V}| \times |\mathcal{Q}|)$ iterations, which can be simplified into $O(|\mathcal{Q}|^3)$. Forming a rewriting takes $O(n_{\mathcal{Q}})$, bringing the total to $O(|\mathcal{Q}|^3 \times n_{\mathcal{Q}})$.

Algorithm 13 needs to compute the following values, which must be estimated *before* any query or view results are computed. We do this as follows:

1. To compute $cost(\cdot)$, the cost to evaluate a query q or materialize a view v, we use the *cost estimation* of the graph data management system (GDBM, in short). Our implementation relies on PostgreSQL, whose `explain` command returns both the estimated number of results of a certain query (or view), denoted $size(q)$, and the cost of computing those results.

2. For $r_{q,v}$, the rewriting of q using a view v, recall that when v is used to rewrite q, v is a sub-path of q, thus there exist i, j such that $v = q|^{i,j}$ (Section 7.4.2). The rewriting $r_{q,v}$ is easily obtained by replacing, in the body of q, the atoms from the i^{th} to the j^{th}, with the head of v.

3. Estimating $cost(r_{q,v})$, the cost of such a rewriting is more complex than evaluating $cost(\cdot)$. This is because the cost of a query (or view) is estimated based on *statistics* the GDBM has about the stored graph. In contrast, the GDBM cannot estimate cost of $r_{q,v}$, because v *has not been materialized* yet, thus the GDBM cannot reason about v like it does about the graph. To compensate, we proceed as follows: we compute the cost of reading the hypothetical view v_{max} from the database, by multiplying $size(v_{max})$, the estimation of the view size, with a constant (we used PostgreSQL's own CPU_TUPLE_COST); then, we estimate the cost of $r_{q,v_{max}}$ as this reading cost plus the cost of estimating the parts of q not in v_{max} plus the cost of joining v_{max} with these (one or two) remaining query parts. We estimate the cost of each such join by adding their input sizes, which we then multiply with another (GDBM) constant. This reflects the fact that modern databases feature efficient join algorithms, such as memory-based hash joins, whose complexity is linear in the size of their inputs.

Regarding our heuristics, (H1) is universally adopted in the literature: no candidate view features cartesian products. (H2), imposing that views benefit at least two queries, preserves result quality, i.e., cost savings, under every *monotone cost model*, ensuring that the cost of evaluating a query q is at least that of evaluating s, when s is a sub-query of q. In contrast, (H3) and (H4) may each divert from the globally optimal solution. However, as our experiments show, our chosen rewritings perform well in practice, and the algorithm itself is very efficient.

7.5 Experimental evaluation

Our approach is fully implemented in a system called PathWays, in Java 11. PathWays relies on ConnectionLens [12, 13] for the construction of the data graph (Chapter 4) and on Abstra [23, 24] which builds the collection graph (Chapter 5); these are stored in PostgreSQL. We experimented

Dataset name	$\|N\|$	$\|E\|$	$\|\tau_P\|$	$\|\tau_L\|$	$\|\tau_O\|$	$min(e_s)$
PubMed	63,052	89,710	5,993	2,151	5,096	0.001
Nasa	59,408	128,068	634	690	4,530	0.0002
YelpBusiness	57,963	61,627	322	427	1,437	0.001
YelpBusiness4	229,949	247,074	1,099	1,230	4,199	0.0002

Table 7.2: Dataset overview.

on a Linux server with an Intel Xeon Gold 5218 CPU @ 2.30 GHz and 196GB of RAM. We used PostgreSQL v9.6. Our evaluation seeks to answer the two following questions:

1. *How does the ChatGPT-based extractor compare with the previous best one available in* ConnectionLens*?* (Section 7.5.2);
2. *How are NEs connected in each dataset?* (Section 7.5.3);
3. *How efficient is our multi-query optimization algorithm in reducing the time to evaluate paths queries over the data graph?* (Section 7.5.4)
4. *How do reliability and force vary in our datasets?* (Section 7.5.5);
5. *How efficient is our path evaluation on top-ranked paths*? (Section 7.5.6).

7.5.1 Datasets and settings

We used three of the datasets we used in Abstra's experimental evaluation, namely PubMed, Nasa and YelpBusiness. Moreover, we also used YelpBusiness4, a dataset 4 times larger than YelpBusiness, to study the scalability of our algorithm. They all come from real-life applications (as opposed to synthetic) to stay close to application needs, and to ensure realistic Named Entities (NEs). Indeed, synthetic datasets are often generated with an interest on structure, while the leaf (text) values lack interesting information. Table 7.2 shows for each dataset: its number of nodes $|N|$, edges $|E|$, numbers of extracted NEs $|\tau_P|$, $|\tau_L|$, $|\tau_O|$ and the minimum edge specificity $min(e_s)$. Without loss of generality, we experiment with the NE types Person, Location, Organization, whose types are denoted τ_P, τ_L, τ_O, respectively. We set L_{max} to 10.

7.5.2 Performance of ChatGPT entity extraction

We have analyzed the novel ChatGPT-based NE extractor introduced in this work (Section 7.2), from the angle of: speed (it is a remotely provided service, whose invocation requires remote calls), financial costs incurred, and results quality.

Table 7.3 shows, for 6 strings taken from the experimental datasets (Section 7.5.1): $|s|$, the number of characters in the string; T_{extr}^{Flair}, resp. T_{extr}^{GPT}, the Flair, resp. ChatGPT, time (in seconds) to send the extraction query, wait for the service answer, and retrieve it through a `java.net.HttpURLConnection`; $|Flair|$, resp. $|GPT|$, the number of NEs found by each extractor in the string; $|t_{cont}|$, the number of context tokens in the ChatGPT prompt; and $|t_{gen}|$, the number of tokens in the ChatGPT output.

With respect to **speed**, we note that Flair extraction time increases with regards to the input string size. On the other side, ChatGPT brings a comparatively huge overhead *per connection* (or per

String	$\|s\|$	T_{extr}^{Flair}	$\|Flair\|$	T_{extr}^{GPT}	$\|GPT\|$	$\|t_{cont}\|$	$\|t_{gen}\|$
s_1	13	0.022	1	2.465	1	126	37
s_2	16	0.020	1	2.208	1	128	40
s_3	80	0.067	3	5.171	5	140	119
s_4	125	0.106	5	6.503	6	148	114
s_5	673	0.371	8	10.741	9	267	194
s_6	3,191	1.931	10	6.789	8	809	176

Table 7.3: Flair and ChatGPT-based extractors time (in seconds) and cost analysis on sample strings.

	GPT Person	GPT Location	GPT Organization	GPT no entity
Flair Person	**5913**	6	11	98
Flair Location	25	**1088**	507	905
Flair Organization	36	141	**2988**	1797
Flair no entity	101	1335	1233	–

Table 7.4: Comparison of Flair and ChatGPT sets of extracted entities.

string sent to the extractor). Thus, T_{extr}^{GPT} times are much higher than T_{extr}^{Flair}, even if extraction itself is not longer than with Flair (which can be tested by sending the same query to the ChatGPT free online assistant). While T_{extr}^{GPT} will vary depending on the caller's internet connection and many other factors hard or impossible to control (where is the actual ChatGPT server located, its load, etc.), the slowdown incurred by the remote connection is likely to occur in all settings.

For what concerns **financial costs** (non-existent for our Flair extractor), GPT-4 use incurs around \$0.03/1K context (or prompt) token and \$0.06/1K generated tokens according to OpenAI[4]. In total, extractions in Table 7.3 used 1,618 context tokens and 680 generated tokens, leading to a cost of \$0.08.

To compare **extraction quality**, Table 7.4 quantifies agreement and disagreement between the Flair and ChatGPT extractors, for the PubMed dataset. Specifically, for each string s found in the data, and entity type τ, each extractor finds a set of entities of type τ. Let's call these sets E_τ^1, E_τ^2.

1. If one set is empty, the size of the other set adds to the counter in the respective τ/"no entity" cell.
2. Otherwise:
   (a) Each entity in $E_\tau^1 \cap E_\tau^2$ counts as 1 in the τ/τ cell;
   (b) Each entity in $(E_\tau^1 \setminus E_\tau^2) \cup (E_\tau^2 \setminus E_\tau^1)$ which the other extractor found with the same label, but of another type $\tau' \neq \tau$, counts as 1 in the cell corresponding to τ for the first extractor and τ' for the other;
   (c) Each entity in $(E_\tau^1 \setminus E_\tau^2) \cup (E_\tau^2 \setminus E_\tau^1)$ such that the other extractor has found no entity (of any type) with the same label in s, adds to the counter in the respective τ/"no entity" cell.

[4]https://openai.com/pricing#language-models

Note that this measure of agreement is conservative, in the following sense: we require identical labels and identical types for entities to be in agreement. If the two extractors find in the same string, respectively, entities e and e', it will show in τ/"no entity" cells (Table 7.4), even if their labels are very close, as in "Lyon" and "Lyon Cedex". Moreover, if e and e' also have the same type, it will also be counted in τ/"no entity" cells of Table 7.4. Observe that *for our purposes, such cases lead to the exact same collection paths being enumerated*, and the same path reliability values. Thus, we do not consider them further.

Table 7.4 shows that **agreement is significant** (entity numbers in bold on the diagonal), and **very frequent for Person** entities. Also, entities recognized as Person by one extractor and of another type by the other are very rare. **ChatGPT disagrees more strongly** with Flair over **Flair's Organizations**, frequently considering that no entity at all exists with the same label. The same holds about **ChatGPT's Locations**: Flair finds no entity with the same label, almost as often as it agrees with ChatGPT. Such cases, when Flair finds an entity and ChatGPT does not, reflect: on one hand, exactly Flair's false positives that we seek to avoid, in order to increase edge and path reliability; on the other hand, extractor disagreements on the tokens to include in an entity label, as we exemplify below.

Flair Person entities not found by ChatGPT include "Claudin-7b", "Cytochrome" and "Claudin-h", which are all proteins. Flair was likely confused by the capitalization, and wrongly extracts them as Person from paper titles and abstracts. Other mistakenly extracted Person entities include people names in street names such as $\underbrace{\text{Peter Henry Rolfs}}_{\text{PERS}}$ in *"Av. Peter Henry Rolfs, 36570-900 Viçosa"*. These mistaken Flair extractions are made on paper titles and author affiliations, leading to unreliable extraction edges in the collection graph, and thus, to unreliable paths. ChatGPT's better training gives it an advantage here.

Flair Location entities not found by ChatGPT are mostly due to different allocations of tokens in entity labels. For instance, in the string "*Institute for Cancer Outcomes and Survivorship, University of Alabama at Birmingham, Birmingham, Alabama, USA*", Flair identifies: $\underbrace{\text{Institute for Cancer Outcomes and Survivorship}}_{\text{ORG}}$, $\underbrace{\text{University of Alabama}}_{\text{ORG}}$, $\underbrace{\text{Birmingham}}_{\text{LOC}}$, $\underbrace{\text{Alabama}}_{\text{LOC}}$ and $\underbrace{\text{USA}}_{\text{LOC}}$ (five entities), while ChatGPT extracts four: $\underbrace{\text{Institute for Cancer Outcomes and Survivorship}}_{\text{ORG}}$, $\underbrace{\text{University of Alabama at Birmingham}}_{\text{ORG}}$, $\underbrace{\text{Birmingham, Alabama}}_{\text{LOC}}$, and $\underbrace{\text{USA}}_{\text{LOC}}$. In our table, this leads to "Alabama" and "Birmingham" counting as two Flair Locations not found by ChatGPT (and symmetrically, "Alabama, Birmingham" counts as a ChatGPT Location not found by Flair). Due to the presence of many affiliation strings in our dataset, such disagreements are frequent in the table cells involving Locations and/or Organizations. In these cases, we found that ChatGPT's choice of entity labels is better. For instance, "Birmingham, Alabama" is more specific than "Birmingham" (the latter exists in many places, among them also UK, etc.). Other Flair-extracted Locations are clearly incorrect, e.g., from "*Av. Professor Egaz Moniz, Lisboa 1649-028, Portugal*", it extracts $\underbrace{\text{Av.}}_{\text{LOC}}$.

Flair Organization entities not found by ChatGPT are mostly due to similar errors (Loca-

tions and Organizations competing for tokens). They also include other obvious errors, e.g., Ag (ORG) (silver), Critique of the Literature (ORG), Lolium perenne L. (ORG) (a plant), Drs. (ORG) (doctors), etc.

ChatGPT Person entities not found by Flair include, e.g., *"Antonio González"*, (Spanish name, probably under-represented in Flair's training set), *"John A. Reif, Jr"* (full name plus the "Jr" suffix, probably also rare in the training set), etc.

ChatGPT Location entities not found by Flair include: "Varese, Italy" (Flair found "Varese" and separately "Italy", see discussion of "Birmingham, Alabama" above), "3-5-7 Tarumi" (address without the city, Japanese format), and numerous addresses including zip codes. ChatGPT has better knowledge of international addresses, e.g., correctly identifying a Location in: the Korean address *"San 65, Bokjeong-Dong, Sujeong-Gu, Seongnam City, Gyeonggi-do, 461-701, South Korea"*; the Japanese address *"Yoshida, Sakyo-ku, Kyoto 606-8501, Japan"*; and the Polish address "*ul. Niezapominajek 8, 30-239 Krakow, Poland*", where Flair only accepted the city and/or the country.

ChatGPT Organization entities not found by Flair are again mostly due to different allocations of tokens in entities. For instance, in *"Oxford Radcliffe Hospitals NHS Trust, Department of Otolaryngology - Head and Neck Surgery, Level LG1, West Wing, John Radcliffe Hospital, Oxford, UK, OX3 9DU."*, ChatGPT finds Department of Otolaryngology - Head and Neck Surgery (ORG) (among others), whereas Flair finds: Department of Otolaryngology (ORG) and Head and Neck Surgery (ORG), two entities instead of the correct single one.

NE type disagreements between extractors The most frequent class of such disagreements (Table 7.4) are Flair Locations considered Organizations by ChatGPT. In these cases, ChatGPT is right: the entity is really an Organization. Some of them include a Location element, e.g., "Middle East Technical University", "The University of Tokyo", "Taipei Medical University", "McGill University", which may have confused Flair, but others do not, e.g., "INRA", "LIFE Center", "Joint Orthopaedic Centre". Conversely, Flair Organizations which ChatGPT considers Locations include: "Lille", "Viet Nam", "Rua de Universidade" (ChatGPT is right; this is a majority of cases), and a handful of cases where Flair is right, e.g., "Ospedale di Busto Arsizio", "Intensive Care Unit". 90% of the Flair Locations and Flair Organizations which GPT finds to be Person entities are initials, such as "A.R.A", "R.H.S", or acronyms such as "M.D" and "PhD". The latter are author academic titles; both extractors are wrong here. The former correspond to authors' initials, found in the paper metadata (conflict of interest statements). ChatGPT is right on "Chiharu Uno" and "L. Giampiero Mazzaglia" (these are people indeed); it also made a mistake on "Korean Firefighters", which should rather be an Organization.

Overall, we find ChatGPT leads to better-quality results, and should be preferred whenever one can afford the budget.

7.5.3 Path enumeration

For each dataset and pair of entity types, Table 7.5 reports the number of paths of each directionality (Section 7.3.2), the minimum and maximum length L_p of each path, and the minimum

	(τ_1, τ_2)	N_{root}	N_{gen}	$\min(L_p)$	$\max(L_p)$	$\min(S_p)$	$\max(S_p)$
PubMed	(τ_P, τ_O)	21	-	5	8	0	13,988
	(τ_P, τ_L)	21	-	5	8	0	15,181
	(τ_L, τ_O)	21	-	5	8	0	5,054
	(τ_P, τ_P)	21	-	5	8	0	389
	(τ_L, τ_L)	21	-	5	8	0	1,214
	(τ_O, τ_O)	21	-	5	8	3	3,090
Nasa	(τ_P, τ_O)	99	1	5	9	0	629
	(τ_P, τ_L)	95	5	5	9	0	137
	(τ_L, τ_O)	97	3	5	9	0	603
	(τ_P, τ_P)	97	3	5	9	0	89
	(τ_L, τ_L)	97	3	5	9	0	3,050
	(τ_O, τ_O)	97	3	5	9	0	8,960
YelpBusiness	(τ_P, τ_O)	41	-	5	7	0	651
	(τ_P, τ_L)	33	-	5	7	0	193
	(τ_L, τ_O)	21	-	5	5	0	1,412
	(τ_P, τ_P)	28	-	5	7	0	35
	(τ_L, τ_L)	15	-	5	5	2	158
	(τ_O, τ_O)	21	-	5	5	0	1,232
YelpBusiness4	(τ_P, τ_O)	48	-	5	7	0	2,593
	(τ_P, τ_L)	39	-	5	7	0	760
	(τ_L, τ_O)	39	-	5	5	0	258
	(τ_P, τ_P)	36	-	5	7	0	207
	(τ_L, τ_L)	15	-	5	5	0	674
	(τ_O, τ_O)	21	-	5	5	0	4,889

Table 7.5: Entity paths found in our datasets and their associated statistics.

and maximum data path support (number of results when evaluated on the data), this is denoted S_p. For the PubMed (XML) and YelpBusiness (JSON) datasets, we obtained only shared-root paths: this is because of the tree structure of these datasets, where text values (leaves) are only connected by going through a common ancestor node. In the RDF Nasa dataset, we also found general-directionality paths. The JSON datasets are more irregular, leading to more paths. In almost every case, a few collection paths had 0 support, due to dataset summarization (Section 7.3). The maximum support may be high, e.g., 15,181 in the PubMed dataset.

These results show that numerous interesting entity paths exist in our datasets, of significant length (up to 9), and some with high support, bringing the need for an efficient evaluation method.

7.5.4 Efficiency of path evaluation

We now study the efficiency of data path computations over the graph. Table 7.6 shows, for each dataset and entity type pair, T_0 is the time to evaluate the corresponding path queries without the view-based optimization of Sections 7.4.2 and 7.4.3, referred to as **VBO** from now on. $|\mathcal{Q}_{TO}|$ is the number of queries whose execution we stopped (time-out of 30s) without VBO. $|\mathcal{Q}_{NV}|$ is the

	(τ_1, τ_2)	T_0	$\lvert\mathcal{Q}_{TO}\rvert$	$\lvert\mathcal{Q}_{NV}\rvert$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{M}\rvert$	T_R	$T_{\mathcal{Q}_{NV}}$	$T{=}T_R{+}T_{\mathcal{Q}_{NV}}$	$s{=}T_0/T$
PubMed	(τ_P, τ_O)	250.36	5	1	16	5	3.78	0.32	4.10	61×
	(τ_P, τ_L)	37.29	0	1	16	5	19.06	0.32	19.38	2×
	(τ_L, τ_O)	151.29	2	2	16	5	11.88	8.59	20.47	7×
	(τ_P, τ_P)	152.59	3	1	16	5	44.19	0.08	44.27	3×
	(τ_L, τ_L)	169.64	2	1	16	5	71.32	0.31	71.63	2×
	(τ_O, τ_O)	317.92	5	1	16	5	22.99	0.25	23.24	13×
Nasa	(τ_P, τ_O)	195.47	1	0	80	10	54.14	-	54.14	3×
	(τ_P, τ_L)	254.26	3	0	68	10	44.57	-	44.57	5×
	(τ_L, τ_O)	1073.55	32	0	77	9	131.58	-	131.58	8×
	(τ_P, τ_P)	278.95	4	0	76	10	92.01	-	92.01	3×
	(τ_L, τ_L)	1103.48	30	0	77	9	101.35	-	101.35	10×
	(τ_O, τ_O)	1318.78	37	0	77	9	247.43	-	247.43	5×
YelpBusiness	(τ_P, τ_O)	205.95	2	0	22	6	4.20	-	4.20	49×
	(τ_P, τ_L)	410.87	7	1	19	5	40.87	1.27	42.12	9×
	(τ_L, τ_O)	239.90	0	1	20	10	1.15	0.6	1.75	137×
	(τ_P, τ_P)	466.58	9	2	23	5	15.33	12.02	27.35	17×
	(τ_L, τ_L)	450.00	15	1	8	4	9.89	< 0.01	9.89	45×
	(τ_O, τ_O)	334.22	4	1	10	5	2.83	< 0.01	2.83	118×
YelpBusiness4	(τ_P, τ_O)	804.70	26	0	23	6	62.52	-	62.52	12×
	(τ_P, τ_L)	454.19	10	1	20	5	92.50	< 0.01	92.50	5×
	(τ_L, τ_O)	242.57	5	1	10	5	62.74	6.61	69.35	3×
	(τ_P, τ_P)	317.00	7	1	27	7	14.35	1.08	15.43	20×
	(τ_L, τ_L)	395.49	10	1	8	4	2.62	18.15	20.77	19×
	(τ_O, τ_O)	347.23	8	1	10	5	42.93	2.34	45.27	7×

Table 7.6: View-based data path evaluation.

	(τ_1, τ_2)	min p_{rel}	max p_{rel}	p_{rel}^{20}	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{P}'\rvert$	$R = \frac{\lvert\mathcal{P}'\rvert}{\lvert\mathcal{P}\rvert}$
PubMed	(τ_P, τ_O)	0.0150	0.9142	0.0409	52	20	38.45%
	(τ_P, τ_L)	0.0150	0.9107	0.0150	30	20	66.66%
	(τ_L, τ_O)	0.0150	0.9107	0.0232	34	20	58.82%
	(τ_P, τ_P)	0.0150	0.9774	0.0150	24	20	83.33%
	(τ_O, τ_O)	0.0150	0.4158	0.0232	31	20	64.51%
	(τ_L, τ_L)	0.0150	0.0954	0.0150	20	20	100.00%
Nasa	(τ_P, τ_O)	0.0014	0.0645	0.0178	191	20	10.47%
	(τ_P, τ_L)	0.0014	0.0645	0.0077	142	20	14.08%
	(τ_L, τ_O)	0.0014	0.1016	0.0077	115	20	17.39%
	(τ_P, τ_P)	0.0014	0.0232	0.0077	110	20	18.18%
	(τ_O, τ_O)	0.0014	0.0581	0.0077	92	20	21.73%
	(τ_L, τ_L)	0.0014	0.3790	0.0077	67	20	29.85%
Yelp	(τ_L, τ_O)	0.0002	0.9997	0.0002	8	8	100.00%
	(τ_L, τ_L)	0.0002	1.0000	0.0002	11	11	100.00%

Table 7.7: Numbers of paths and reliability information in our datasets.

number of queries for which Algorithm 13 did not recommend a view. T_R is the time to evaluate the rewritten queries on the data graph, while $T_{\mathcal{Q}_{NV}}$ is the time to evaluate the non-rewritten queries $\mathcal{Q}_{NV}$; $T = T_R + T_{\mathcal{Q}_{NV}}$ is the (total) execution time to evaluate queries using VBO. Finally, $s = T_0/T$ is the speed-up thanks to VBO. We do not report times to materialize views because they were all very short (less than 0.01s). All times are in seconds.

The evaluation time T_0 without VBO ranges from 37s to 1318s; these path queries require 5 to 9 joins, on graphs of up to more than 250,000 edges (Table 7.2). $|\mathcal{Q}_{NV}|$, the number of queries that could not make use of any views, is rather small, which is good. The number of candidate views, respectively, materialized views depend on the complexity of the dataset, and thus on the complexity of the paths. The total path evaluation time T is reasonable. Finally, the VBO speed-up is at least 2× and at most 137×, showing that our view-based algorithm allows to evaluate path queries much more efficiently.

7.5.5 Path reliability and ranking

We now study how reliability (Section 7.3.3) and force (Section 7.3.4) vary across paths. Table 7.7 illustrates reliability of enumerated paths on our datasets, loaded with the ChatGPT-based NE extractor (Section 7.2). We excluded YelpBusiness4 in order to limit ChatGPT expenses. For each dataset and a pair of NE types connected by at least a path in a collection graph, we show: min p_{rel}, the minimal path reliability among all the paths connecting NEs of these types; max p_{rel}, the maximal path reliability for the same paths; p_{rel}^{20}, the reliability of the 20th ranked path (20 is the default number of paths shown to the user for each pair of entity types); $|\mathcal{P}|$, the number of enumerated paths; $|\mathcal{P}'|$, which is either 20 if there are at least 20 paths, or $|\mathcal{P}|$ otherwise. We also show the ratio R between $|\mathcal{P}'|$ and $|\mathcal{P}|$. Path reliability values span over the whole $(0, 1]$ interval, e.g., 1.000 or 0.9997 in the YelpBusiness dataset, or 0.9774 in PubMed, to 0.0002 for some in YelpBusiness. Thus, reliability gives a strong signal for ranking paths.

Table 7.8 illustrates some paths between τ_P and τ_O, *ordered by reliability, then force*, in the PubMed

p^{id}	p_{rel}	$F(p)$	Path p connecting a Person (τ_P) entity with an Organization (τ_O) entity
p^1	0.91	1.00	$\tau_P \leftarrow_{1.0}$ #val $\leftarrow$ Name $\leftarrow$ Author $\rightarrow$ Affiliation $\rightarrow$ #val $\rightarrow_{0.91}$ τ_O
p^2	0.41	0.02	$\tau_P \leftarrow_{1.0}$ #val $\leftarrow$ Name $\leftarrow$ Author $\leftarrow_{0.02}$ AuthorList $\leftarrow$ PubmedArticle $\rightarrow$ JournalTitle $\rightarrow$ #val $\rightarrow_{0.41}$ τ_O
p^3	0.09	1.00	$\tau_P \leftarrow_{0.09}$ #val $\leftarrow$ CoiStatement $\leftarrow$ PubmedArticle $\rightarrow$ JournalTitle $\rightarrow$ #val $\rightarrow_{0.41}$ τ_O
p^4	0.09	0.02	$\tau_P \leftarrow_{0.09}$ #val $\leftarrow$ CoiStatement $\leftarrow$ PubmedArticle $\rightarrow$ AuthorList $\rightarrow_{0.02}$ Author $\rightarrow$ Affiliation $\rightarrow$ #val $\rightarrow_{0.91}$ τ_O
...	...	...	...
p^{20}	0.04	1.0	$\tau_P \leftarrow_{1.0}$ #val $\leftarrow$ Name $\leftarrow$ Author $\rightarrow$ Affiliation $\rightarrow$ #val $\rightarrow_{0.91}$ $\tau_L \leftarrow_{0.09}$ #val $\rightarrow_{0.04}$ τ_O
p^{21}	0.04	1.00	$\tau_P \leftarrow_{0.09}$ #val $\leftarrow$ CoiStatement $\leftarrow$ PubmedArticle $\rightarrow$ JournalTitle $\rightarrow$ #val $\rightarrow_{0.05}$ $\tau_L \leftarrow_{0.09}$ #val $\rightarrow_{0.04}$ τ_O
p^{22}	0.04	0.02	$\tau_P \leftarrow_{1.0}$ #val $\leftarrow$ Name $\leftarrow$ Author $\leftarrow_{0.02}$ AuthorList $\leftarrow$ PubmedArticle $\rightarrow$ ArticleTitle $\rightarrow$ #val $\rightarrow_{0.04}$ τ_O

Table 7.8: Some of the top-reliability (τ_P, τ_O) paths in the PubMed dataset, at ranks: 1, 2, 3, 4, 20 (above the double line), respectively, 21 and 22 (below the double line), out of 52 paths.

dataset; we picked these since this is the largest group of paths (52 initially, recall Table 7.7). It features the first four and the last paths among the top-ranked paths, as well as p^{21} and p^{22}, which almost made it to the top-20. For each of them, Table 7.8 shows the path reliability, its force and the collections it connects. We show the reliability of each extraction edge as an index, and similarly, show the force of each non-extraction edge when it is smaller than 1. For instance, the edge connecting the XML element ⟨AuthorList⟩ to all its ⟨Author⟩ children has a force of 0.02. The most reliable path connects authors with the organizations to which they are affiliated. The second connects authors to organizations present in the title of the journals where their articles appear, such as $\underbrace{\text{American Chemical Society}}_{ORG}$ identified in *"Journal of the American Chemical Society"*. The following paths connect, respectively: people mentioned in the paper's Conflict of Interest statements with organizations appearing in journal titles (third), and author's employers (fourth), respectively. p^{20} connects an author's name to its affiliation location, such that this affiliation also contains a location found in journal titles. p^{21} connects people found in conflict of interest statements to organizations found in article titles, to which a low reliability of 0.04 is given. p^{22} connects an author's name to the organization that may be found in an article title. This path has a low reliability (only 0.04) and its force is also low (0.02), due to the many authors that an article may have.

The least reliable collection extraction edges found in PubMed include: 6 Locations extracted from 399 CoiStatement values (leading to a reliability of 0.015) and 86 people names in 5712 Affiliations (reliability also equal to 0.015). They were found in strings such as "$\underbrace{\text{James J. Peters}}_{PERS}$ VA Medical Center, 130 West Kingsbridge Road, Bronx, NY 10468, USA.". All paths traversing this edge are at the very bottom of the ranked list of paths. Analysis of path reliability in the other datasets lead to similar findings.

(τ_1, τ_2)	T_0	$\lvert\mathcal{Q}_{TO}\rvert$	$\lvert\mathcal{Q}_{NV}\rvert$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{M}\rvert$	T_R	$T_{\mathcal{Q}_{NV}}$	T=T_R+$T_{\mathcal{Q}_{NV}}$	s=T_0/T
PubMed									
(τ_P, τ_O)	6.29	0	1	32	4	1.80	0.04	1.84	3.4×
(τ_P, τ_L)	38.73	1	1	26	4	1.57	0.02	1.59	24.3×
(τ_L, τ_O)	68.80	2	1	24	4	4.04	0.02	4.06	16.9×
(τ_P, τ_P)	2.21	0	4	26	5	9.08	0.04	9.12	0.2×
(τ_O, τ_O)	20.20	0	1	33	4	18.65	0.01	18.66	1.1×
(τ_L, τ_L)	23.57	0	7	16	2	1.38	0.01	1.39	16.9×
Nasa									
(τ_P, τ_O)	11.30	0	1	46	2	10.04	8.82	18.86	0.5×
(τ_P, τ_L)	29.77	0	1	41	4	2.37	0.03	2.40	12.4×
(τ_L, τ_O)	36.57	1	1	29	4	4.71	0.03	4.74	7.7×
(τ_P, τ_P)	3.13	0	1	35	2	2.46	0.01	2.47	1.2×
(τ_O, τ_O)	32.66	1	3	35	4	2.89	0.04	2.93	11.1×
(τ_L, τ_L)	36.13	0	6	19	2	1.62	0.09	1.71	21.1×
YelpBusiness									
(τ_L, τ_O)	38.86	1	0	7	1	3.62	0	3.62	10.7×
(τ_L, τ_L)	131.98	4	2	6	3	45.74	0.15	45.89	2.8×

Table 7.9: Data path evaluation on the top-20 enumerated paths, sorted by reliability, then force, in the PubMed, Nasa and YelpBusiness datasets.

7.5.6 Evaluation of the top-ranked paths

Table 7.9 studies the benefit of the MQO approach (Algorithm 13) on the top-ranked paths. For each dataset ingested with the ChatGPT-based NE extractor, we show the same information as presented in Table 7.6 (Section 7.5.4). All the evaluation times, notably T_0 when directly evaluating paths, and T when applying our MQO, are given in seconds. The speed-up (rightmost column) shows that for 12 out of 14 path groups, MQO achieves its goal, reducing evaluation times by up to 10× or 20×. We also notice two path groups whose evaluation it slowed by MQO, by a factor of 2×, respectively, 5×. This is due to its heuristic choices and relatively simple cost model. The gains are slightly less than those in Table 7.6; this may be because there were in general more paths in that table, leading to more sharing opportunities. Still, we find the MQO gains are generally robust and significant, confirming its interest.

7.5.7 Experiment conclusion

Our experiments lead to the following observations. First, the ChatGPT entity extractor improved the overall path quality by (*i*) a better recognition of Locations, at all geographical levels (street up to country); (*ii*) a significantly better recognition of Organizations, avoiding false positives and finding good entity labels; (*iii*) modest improvements in the quality of extracting people names; and (*iv*) overall, a better support of many languages (Section 7.5.2). All these advantages can be attributed to its large training corpus, and we recommend it whenever extraction time is not crucial, and financial costs can be afforded. Second, many Named Entity paths exist, in the datasets we considered (Section 7.5.3). Third, our path evaluation algorithm is very effective in reducing the time to evaluate path sets (Section 7.5.4). Fourth, our novel ranking based on reliability and force

downgrades many meaningless paths, while preserving significant ones (Section 7.5.5). Our joint path materialization technique is effective also on the top-ranked paths (Section 7.5.6).

7.6 Summary

In this chapter, we have presented how to efficiently enumerate and evaluate a set of interesting paths in and across datasets, leveraging the data graph (Chapter 4) and the collection graph (Chapter 5). We addressed two main challenges, i.e., identify interesting entity-to-entity paths and efficiently find them in a data graph.

We tackled the first challenge by designing a high-quality Named Entity extractor using ChatGPT 4 and ranked paths based on their interestingness, which we define as a combination of extraction reliability and information force. Our experiments have shown that our new NE extractor and interestingness measure allow to show users meaningful paths. For the second challenge, paths are efficiently enumerated using the collection graph, even if the data is large and complex. Enumerated paths are evaluated on the data graph, leveraging the idea they may share materializable sub-paths. Experiments have shown that this allows important speed-ups.

8

Conclusions and perspectives

In this thesis, we have provided novel means to help users and data providers understand, explore, and share heterogeneous semi-structured datasets they may work with. Working with semi-structured data is challenging, especially for non-technical users, and especially when multiple data models must be handled simultaneously. This thesis has proposed methods and tools working on any structured and semi-structured data model, i.e., relational data, XML documents, JSON documents, RDF graphs and Property graphs. This has led to the following proposals:

- A unified view over how datasets model objects (*records*), relationships between objects, and objects of the same *kind*.
- The notion of *same-kind* records, allowing to group records, not only based on their (explicit) type, but also on their implicit kind.
- The *collection graph*, a core structure in our works, allowing to work on a summarized version of the data.

Next, we have introduced novel *data abstractions*: compact, but expressive, descriptions of data in the form of Entity-Relationship schemas, potentially nested to accommodate semi-structured data complexity. For this, we proposed Abstra, a method and software which:

- Automatically identifies, in a given dataset, a set of main entities and their relationships; they broadly correspond to the entities one could create when designing a relational database.
- Determines which collections in the collection graph deserve to be elected as main entities, using various weight schemes, some simple, some more intricated, based on PageRank. Further, we identify the (potentially nested) properties belonging to each main entity, included in the *entity boundaries*.
- Classifies each main entity among a set of categories, that helps the overall understanding of the dataset, especially if users are not familiar with the dataset application domain.
- Is scalable to large datasets, as shown in the experimental evaluation.

Perspectives and open questions Numerous intriguing avenues exist for future work. We highlight some, that could be immediately pursued.

Our detection of relationships is only capable of finding binary relationships (between two entities). It would be interesting to extend this approach in order to capture also relations of higher arity. For that, data profiling could be used, also taking inspiration from the data model where n-ary relations are represented in well-known ways.

A different avenue of research leverages the vision presented in this thesis in order to convert data of various data models, into Property Graphs. The latter model is the target of many research activities today, on the data model, query language, efficient query processing, in particular in-memory, etc. Many interesting datasets are available in data models that are not natively PGs. We have started a follow-up project (not reported in the manuscript) where, from the data abstraction, we compute a PG schema. The next logical step, continuing on this line, is to migrate (map) also the data, from the normalized data graph created by Abstra, into the PG format.

An orthogonal question concerns the construction (or enrichment) of a given dataset. Once loaded in ConnectionLens and abstracted by Abstra, existing Open Data sources could be leveraged to augment the dataset with external knowledge about the Named Entities it contains. We have started work along this direction (not reported in the manuscript), leveraging RDF Knowledge Bases (KBs).

It would be intriguing also to further experiment the usage of abstractions in very large data lakes, with thousands or tens of thousands of datasets. On one hand, to speed up dataset abstraction, one could resort to sampling. On the other hand, once abstractions have been computed, it would be interesting to understand how to aggregate or search over them, in order to help users quickly identify the datasets that could be most useful to them.

Bibliography

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. *Data Profiling*. Morgan & Claypool Publishers, 2018.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[3] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web data management*. Cambridge University Press, 2011.

[4] B Aditya, Gaurav Bhalotia, Soumen Chakrabarti, Arvind Hulgeri, Charuta Nakhe, S Sudarshanxe, et al. BANKS: browsing and keyword searching in relational databases. In *VLDB*, pages 1083–1086. Elsevier, 2002.

[5] Christian Aebeloe, Vinay Setty, Gabriela Montoya, and Katja Hose. Top-K diversification for path queries in knowledge graphs. In *ISWC (Workshops)*, 2018.

[6] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[7] Rafi Ahmed, Philippe De Smedt, Weimin Du, William Kent, Mohammad A. Ketabchi, Witold A. Litwin, Abbas Rafii, and Ming-Chi Shan. The Pegasus heterogeneous multidatabase system. *Computer*, 24(12):19–27, 1991.

[8] Alan Akbik, Tanja Bergmann, Duncan Blythe, Kashif Rasul, Stefan Schweter, and Roland Vollgraf. FLAIR: An easy-to-use framework for state-of-the-art NLP. In *NAACL (demonstrations)*, pages 54–59, 2019.

[9] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. Towards scalable hybrid stores: Constraint-based rewriting to the rescue. In *SIGMOD*, 2019.

[10] Rana Alotaibi, Bogdan Cautis, Alin Deutsch, Moustafa Latrache, Ioana Manolescu, and Yifei Yang. ESTOCADA: towards scalable polystore systems. *PVLDB*, 13(12):2949–2952, 2020.

[11] Ayman Alserafi, Alberto Abelló, Oscar Romero, and Toon Calders. Keeping the data lake in form: DS-kNN datasets categorization using proximity mining. In *MEDI*, pages 35–49. Springer, 2019.

[12] Angelos Anadiotis, Oana Balalau, Théo Bouganim, et al. Empowering investigative journalism with graph-based heterogeneous data management. *IEEE DEBull.*, 2021.

[13] Angelos Anadiotis, Oana Balalau, Catarina Conceicao, et al. Graph integration of structured, semistructured and unstructured data for data journalism. *Information Systems*, 104, 2022.

[14] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. PG-Schema: schemas for property graphs. *SIGMOD*, 1(2):1–25, 2023.

[15] Andrei Arion, Angela Bonifati, Ioana Manolescu, and Andrea Pugliese. Path summaries and path partitioning in modern XML databases. *WWW*, 11(1), 2008.

[16] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta Lake: high-performance ACID table storage over cloud object stores. *PVLDB*, 13(12):3411–3424, 2020.

[17] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *CIDR*, volume 8, 2021.

[18] Sören Auer et al. DBpedia: A nucleus for a web of open data. In *The Semantic Web*, 2007.

[19] Mohamed Amine Baazizi, Clément Berti, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Human-in-the-loop schema inference for massive JSON datasets. In *EDBT*, 2020.

[20] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Parametric schema inference for massive JSON datasets. *The VLDB Journal*, 28(4), 2019.

[21] Zhifeng Bao, Jiaheng Lu, Tok Wang Ling, and Bo Chen. Towards an effective XML keyword search. *TKDE*, 22(8):1077–1092, 2010.

[22] Nelly Barret, Antoine Gauquier, Jia Jean Law, and Ioana Manolescu. Exploring heterogeneous data graphs through their entity paths. In *ADBIS*, volume 13985, pages 163–179. Springer, 2023.

[23] Nelly Barret, Ioana Manolescu, and Prajna Upadhyay. Abstra: toward generic abstractions for data of any model (demonstration). In *CIKM*, 2022.

[24] Nelly Barret, Ioana Manolescu, and Prajna Upadhyay. Computing generic abstractions from application datasets. In *EDBT*, 2024.

[25] Amin Beheshti, Boualem Benatallah, Reza Nouri, Van Munin Chhieng, Huang Tao Xiong, and Xu Zhao. CoreDB: a data lake service. In *CIKM*, pages 2451–2454, 2017.

[26] Domenico Beneventano, Claudio Gennaro, Sonia Bergamaschi, and Fausto Rabitti. A mediator-based approach for integrating heterogeneous multimedia sources. *Multimedia tools and applications*, 62:427–450, 2013.

[27] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *TWEB*, 4(4):1–32, 2010.

[28] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, pages 998–1009. ACM, 2007.

[29] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[30] Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *IJSWIS*, 5(2), 2009.

[31] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. Bridging the XML relational divide with LegoDB. In *ICDE*, 2003.

[32] Angela Bonifati, Stefania Dumbrava, and Haridimos Kondylakis. Graph summarization. *CoRR*, abs/2004.14794, 2020.

[33] Angela Bonifati, Stefania Dumbrava, and Nicolas Mir. Hierarchical clustering for property graph schema discovery. In *EDBT*. OpenProceedings.org, 2022.

[34] Angela Bonifati, Stefania-Gabriela Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacome Luton, and Thomas Pickles. DiscoPG: Property graph schema discovery and exploration. *PVLDB*, 15(12), 2022.

[35] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, 2013.

[36] Liliana Bounegru and Jonathan Gray. *The data journalism handbook: Towards a critical data practice*. Amsterdam University Press, 2021.

[37] Dan Brickley, Matthew Burgess, and Natasha Noy. Google Dataset Search: building a search engine for datasets in an open web ecosystem. In *WWW*, pages 1365–1375, 2019.

[38] David Guy Brizan and Abdullah Uz Tansel. A survey of entity resolution and record linkage methodologies. *Communications of the IIMA*, 6(3):5, 2006.

[39] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Reasoning about keys for XML. *Information Systems*, 28(8):1037–1063, 2003.

[40] Maxime Buron, François Goasdoué, Ioana Manolescu, and Marie-Laure Mugnier. Obi-Wan: ontology-based RDF integration of heterogeneous data. *PVLDB*, 13(12):2933–2936, 2020.

[41] Michael J. Cafarella, Alon Halevy, and Nodira Khoussainova. Data integration for the relational web. *PVLDB*, 2(1):1090–1101, 2009.

[42] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Query processing under GLAV mappings for relational and graph databases. *PVLDB*, 6(2):61–72, 2012.

[43] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. Summarizing semantic graphs: A survey. *The VLDB Journal*, 28(3), 2019.

[44] Camille Chanial, Rédouane Dziri, Helena Galhardas, et al. ConnectionLens: Finding connections across heterogeneous data sources (demonstration). *PVLDB*, 11(12), 2018.

[45] Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. Query recommendations for interactive database exploration. In *SSDBM*, pages 3–18. Springer, 2009.

[46] Wei Chen, Fangzhou Guo, Dongming Han, Jacheng Pan, Xiaotao Nie, Jiazhi Xia, and Xiaolong Zhang. Structure-based suggestive exploration: A new approach for effective exploration of large networks. *IEEE Transactions on Visualization and Computer Graphics*, 25(1), 2019.

[47] Zhiyu Chen, Mohamed Trabelsi, Jeff Heflin, Yinan Xu, and Brian D. Davison. Table search using a deep contextualized language model. In *SIGIR*, pages 589–598, 2020.

[48] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schemas for safe and efficient XML processing. In *ICDE*. IEEE Computer Society, 2011.

[49] Dario Colazzo, François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. RDF analytics: lenses over semantic graphs. In *WWW*, pages 467–478, 2014.

[50] George Colliat. OLAP, relational, and multidimensional database systems. *SIGMOD*, 25(3):64–69, 1996.

[51] Zhamak Dehghani. *Data mesh*. O'Reilly Media, 2022.

[52] Alin Deutsch, Mary F. Fernández, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD*, 1999.

[53] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, et al. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD*, 2022.

[54] Barry A. Devlin and Paul T. Murphy. An architecture for a business and information system. *IBM systems Journal*, 27(1):60–80, 1988.

[55] Gonzalo Diaz, Marcelo Arenas, and Michael Benedikt. SPARQLByE: querying RDF data by example. *PVLDB*, 9(13):1533–1536, 2016.

[56] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-example: an automatic query steering framework for interactive data exploration. In *SIGMOD*, pages 517–528, 2014.

[57] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. In *ICDE*, pages 456–467. IEEE, 2021.

[58] Marina Drosou and Evaggelia Pitoura. YMALDB: exploring relational databases via result-driven recommendations. *The VLDB Journal*, 22(6):849–874, 2013.

[59] Marek Dudáš, Vojtěch Svátek, and Jindřich Mynarz. Dataset summary visualization with LODSight. In *ESWC (Satellite Events)*, pages 36–40. Springer, 2015.

[60] Oliver Michael Duschka. *Query planning and optimization in information integration*. Stanford University, 1998.

[61] Ahmed El-Roby, Khaled Ammar, Ashraf Aboulnaga, and Jimmy Lin. Sapphire: querying RDF data made simple. *arXiv preprint arXiv:1805.11728*, 2018.

[62] Shady Elbassuoni and Roi Blanco. Keyword search over RDF graphs. In *CIKM*, pages 237–242, 2011.

[63] Hassan A. Elmadany, Marco Alfonse, and Mostafa Aref. XML summarization: A survey. In *ICICIS*, pages 537–541, 2015.

[64] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, 2015.

[65] Ronald Fagin, Laura M. Haas, Mauricio Hernández, Renée J. Miller, Lucian Popa, and Yannis Velegrakis. CLIO: schema mapping creation and data exchange. *Conceptual Modeling: Foundations and Applications*, pages 198–236, 2009.

[66] Ju Fan, Guoliang Li, and Lizhu Zhou. Interactive SQL query suggestion: Making databases user-friendly. In *ICDE*, pages 351–362. IEEE, 2011.

[67] Mina Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. CLAMS: bringing quality to data lakes. In *SIGMOD*, pages 2089–2092, 2016.

[68] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. AURUM: A data discovery system. In *ICDE*, pages 1001–1012. IEEE, 2018.

[69] Jenny Rose Finkel, Trond Grenager, and Christopher D. Manning. Incorporating non-local information into information extraction systems by GIBBS sampling. In *ACL*, pages 363–370, 2005.

[70] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into XML query processing. *Computer Networks*, 33(1-6):119–135, 2000.

[71] Kun Fu, Tingyun Mao, Yang Wang, Daoyu Lin, Yuanben Zhang, Junjian Zhan, Xian Sun, and Feng Li. TS-Extractor: large graph exploration via subgraph extraction based on topological and semantic information. *Journal of Visualization*, 24, 2021.

[72] Yihan Gao, Silu Huang, and Aditya Parameswaran. Navigating the data lake with DATAMARAN: Automatically extracting structure from log datasets. In *SIGMOD*, pages 943–958, 2018.

[73] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[74] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. DTD inference from XML documents: The XTRACT approach. *IEEE DEBull.*, 26(3):19–25, 2003.

[75] Corinna Giebler, Christoph Gröger, Eva Hoos, Holger Schwarz, and Bernhard Mitschang. Modeling data lakes with data vault: practical experiences, assessment, and lessons learned. In *ER*, pages 63–77. Springer, 2019.

[76] François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. Incremental structural summarization of RDF graphs. In *EDBT*, 2019.

[77] François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. RDF graph summarization for first-sight structure discovery. *The VLDB Journal*, 29(5), 2020.

[78] François Goasdoué, Ioana Manolescu, and Alexandra Roatis. Efficient query answering against dynamic RDF databases. In *EDBT*. ACM, 2013.

[79] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[80] Alex Gorelik. *The enterprise big data lake: Delivering the promise of big data and data science*. O'Reilly Media, 2019.

[81] Benoît Groz, Aurélien Lemay, Slawek Staworko, and Piotr Wieczorek. Inference of shape graphs for graph databases. In *ICDT*, volume 220, 2022.

[82] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: a benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3), 2005.

[83] Rihan Hai, Sandra Geisler, and Christoph Quix. Constance: an intelligent data lake system. In *SIGMOD*, 2016.

[84] Rihan Hai, Christos Koutras, Christoph Quix, and Matthias Jarke. Data lakes: A survey of functions and systems. *TKDE*, 2023.

[85] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.

[86] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Managing Google's data lake: an overview of the GOODS system. *IEEE DEBull.*, 39(3):5–14, 2016.

[87] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining concepts and techniques, third edition*. 2012.

[88] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[89] Ahmed Helal, Mossad Helali, Khaled Ammar, and Essam Mansour. A demonstration of KGLac: a data discovery and enrichment platform for data science. *PVLDB*, 14(12):2675–2678, 2021.

[90] Tony Hey. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[91] John E. Hopcroft. An nlog n algorithm for minimizing the states in a finite automaton. *The Theory of Machines and Computation*, 1970.

[92] Katja Hose and Ralf Schenkel. Towards benefit-based RDF source selection for SPARQL queries. In *SWIM*, pages 1–8, 2012.

[93] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, pages 670–681. Elsevier, 2002.

[94] Madelon Hulsebos, Çagatay Demiralp, and Paul Groth. GitTables: A large-scale corpus of relational tables. *CoRR*, abs/2106.07258, 2021.

[95] Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çağatay Demiralp, and César A. Hidalgo. Sherlock: a deep learning approach to semantic data type detection. *SIGKDD Explorations*, 2019.

[96] Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Towards exploratory OLAP over linked open data – a case study. In *BIRTE*, pages 114–132. Springer, 2015.

[97] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *SIGMOD*, pages 277–281, 2015.

[98] Lan Jiang and Felix Naumann. Holistic primary key and foreign key detection. *Journal of Intelligent Information Systems*, 54(3), 2020.

[99] Richard M. Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[100] Avita Katal, Mohammad Wazid, and R. H. Goudar. Big data: Issues, challenges, tools and good practices. In *IC3*, pages 404–409, 2013.

[101] Kenza Kellou-Menouer, Nikolaos Kardoulakis, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. A survey on semantic schema discovery. *The VLDB Journal*, 2021.

[102] Kenza Kellou-Menouer and Zoubida Kedad. Schema discovery in RDF data sources. In *ER*. Springer, 2015.

[103] Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Set-based approximate approach for lossless graph summarization. *Computing*, 97:1185–1207, 2015.

[104] Shahan Khatchadourian and Mariano P. Consens. ExpLOD: summary-based exploration of interlinking and RDF usage in the Linked Open Data Cloud. In *ESWC*, pages 272–287. Springer, 2010.

[105] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. SnipSuggest: context-aware autocompletion for SQL. *PVLDB*, 4(1):22–33, 2010.

[106] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. Natural language to SQL: Where are we today? *PVLDB*, 13(10):1737–1750, 2020.

[107] Hyoung-Joo Kim, Henry F. Korth, and Avi Silberschatz. PICASSO: a graphical query language. *Software: Practice and Experience*, 18(3):169–203, 1988.

[108] Ralph Kimball and Margy Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[109] Michal Klempa, Jakub Stárka, and Irena Mlynková. Optimization and refinement of XML schema inference approaches. *Procedia Computer Science*, 10:120–127, 2012.

[110] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[111] Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jiménez-Peris, Raquel Pau, and José Pereira. CloudMdsQL: querying heterogeneous cloud data stores with a common language. *Distributed and parallel databases*, 34:463–503, 2016.

[112] Ora Lassila, Michael Schmidt, Olaf Hartig, Brad Bebee, Dave Bechberger, Willem Broekema, Ankesh Khandelwal, Kelvin Lawrence, Carlos-Manuel López-Enríquez, Ronak Sharda, and Bryan B. Thompson. The OneGraph vision: Challenges of breaking the graph model lock-in. *Semantic Web*, 14(1):125–134, 2023.

[113] Hanâ Lbath, Angela Bonifati, and Russ Harmer. Schema inference for property graphs. In *EDBT*, 2021.

[114] Alon Y. Levy, Anand Rajaraman, Joann J. Ordille, et al. Querying heterogeneous information sources using source descriptions. In *VLDB*, volume 96, pages 251–262. Citeseer, 1996.

[115] Fei Li and Hosagrahar V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

[116] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[117] Matteo Lissandrini, Davide Mottin, Katja Hose, and Torben Bach Pedersen. Knowledge graph exploration systems: are we lost? In *CIDR*, 2022.

[118] Matteo Lissandrini, Davide Mottin, Themis Palpanas, Yannis Velegrakis, and HV Jagadish. *Data Exploration Using Example-Based Methods*. Springer, 2019.

[119] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM Computing Surveys*, 51(3):1–34, 2018.

[120] Ziyang Liu and Yi Chen. Processing keyword search on XML: a survey. *World Wide Web*, 14:671–707, 2011.

[121] Antonio Maccioni and Riccardo Torlone. KAYAK: a framework for just-in-time data preparation in a data lake. In *CAiSE*, pages 474–489. Springer, 2018.

[122] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL (demonstrations)*, pages 55–60, 2014.

[123] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari, and Dan Olteanu. AGORA: living with XML and relational. In *VLDB*, pages 623–626. Citeseer, 2000.

[124] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML schema. *TODS*, 31(3):770–813, 2006.

[125] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[126] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, pages 277–295. Springer, 1999.

[127] Ankur Moitra. *Algorithmic aspects of machine learning*. Cambridge University Press, 2018.

[128] José de Aguiar Moraes Filho. *Summarizing XML Documents: Contributions, Empirical Studies, and Challenges*. PhD thesis, Technische Universität Kaiserslautern, 2010.

[129] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014.

[130] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, Themis Palpanas, et al. New trends on exploratory methods for data analytics. *PVLDB*, 10(12):1977–1980, 2017.

[131] Fatemeh Nargesian, Ken Q. Pu, Erkang Zhu, Bahar Ghadiri Bashardoost, and Renée J. Miller. Organizing data lakes for navigation. In *SIGMOD*, pages 1939–1950, 2020.

[132] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data lake management: challenges and opportunities. *PVLDB*, 12(12):1986–1989, 2019.

[133] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *SIGMOD*, pages 419–432, 2008.

[134] Iuri D. Nogueira, Maram Romdhane, and Jérôme Darmont. Modeling data lake metadata with a data vault. In *IDEAS*, pages 253–261, 2018.

[135] Silvio Normey, Lorena Etcheverry, Adriana Marotta, and Mariano P. Consens. Findings from two decades of research on schema discovery using a systematic literature review. *CEUR (Workshops)*, 2100, 2018.

[136] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[137] Norman W. Paton, Jiaoyan Chen, and Zhenyu Wu. Dataset discovery and exploration: A survey. *ACM Computing Surveys*, 2023.

[138] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian Suchanek. YAGO 4: a reason-able knowledge base. In *ESWC*, 2020.

[139] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *WWW*, pages 263–273, 2016.

[140] Rakesh Pimplikar and Sunita Sarawagi. Answering table queries on the web using column keywords. *arXiv preprint arXiv:1207.0132*, 2012.

[141] Rachel Pottinger and Alon Levy. A scalable algorithm for answering queries using views. In *VLDB*, pages 484–495, 2000.

[142] Valentina Presutti, Lora Aroyo, Alessandro Adamou, Balthasar Schopman, Aldo Gangemi, and Guus Schreiber. Extracting core knowledge from linked data. 2011.

[143] Kashif Rabbani, Matteo Lissandrini, and Katja Hose. Extraction of validating shapes from very large knowledge graphs. *PVLDB*, 2033.

[144] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.

[145] Raghu Ramakhrishnan and Johannes Gehrke. *Database Management Systems (3rd edition)*. McGraw-Hill, 2003.

[146] Chantal Reynaud. Building scalable mediator systems. In *IFIP WCC*, pages 25–30. Springer, 2004.

[147] Matteo Riondato, David García-Soriano, and Francesco Bonchi. Graph summarization with quality guarantees. *Data mining and knowledge discovery*, 31:314–349, 2017.

[148] Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *CTS*, pages 42–47, 2013.

[149] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. ATHENA: an ontology-driven system for natural language querying over relational data stores. *PVLDB*, 9(12):1209–1220, 2016.

[150] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *PVLDB*, 2002.

[151] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*, 2021.

[152] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. ATHENA++: natural language querying for complex nested sql queries. *PVLDB*, 13(12):2747–2759, 2020.

[153] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.

[154] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. In *SIGMOD*, pages 493–504, 2014.

[155] Gary H. Sockut, Luanne M. Burns, Ashok Malhotra, and Kyu-Young Whang. GRAQULA: a graphical query language for entity-relationship or relational databases. *Data & Knowledge Engineering*, 11(2):171–202, 1993.

[156] Qi Song, Yinghui Wu, Peng Lin, Luna Xin Dong, and Hui Sun. Mining summaries for knowledge graph search. *TKDE*, 30(10):1887–1900, 2018.

[157] William Spoth, Oliver A. Kennedy, Ying Lu, Beda Hammerschmidt, and Zhen Hua Liu. Reducing ambiguity in JSON schema discovery. In *SIGMOD*, 2021.

[158] William Spoth, Ting Xie, Oliver Kennedy, Ying Yang, Beda Hammerschmidt, Zhen Hua Liu, and Dieter Gawlick. SchemaDrill: interactive semi-structured schema design. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–7, 2018.

[159] Jannik Strötgen and Michael Gertz. HeidelTime: High quality rule-based extraction and normalization of temporal expressions. In *SemEval@ACL*, pages 321–324, 2010.

[160] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. Enabling query processing across heterogeneous data models: A survey. In *Big Data*, pages 3211–3220, 2017.

[161] Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10), 2014.

[162] RDF Schema. Available online at https://www.w3.org/TR/rdf12-schema/.

[163] Coral Walker and Hassan Alrehamy. Personal data lake with data gravity pull. In *BDCloud*, pages 160–167. IEEE, 2015.

[164] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-SQL: relation-aware schema encoding and linking for text-to-SQL parsers. *arXiv preprint arXiv:1911.04942*, 2019.

[165] Haixun Wang and Charu C. Aggarwal. A survey of algorithms for keyword search on graph data. *Managing and mining graph data*, pages 249–273, 2010.

[166] Gio Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.

[167] Query a Neo4j database using Cypher. Available online at https://neo4j.com/developer/cypher/.

[168] W3C XML Document Type Specification. Available online at https://www.w3.org/TR/REC-xml/#dt-doctype, 2008.

[169] SQL Standards. Available online at https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/SQL-Standards.html.

[170] JSON Schema. Available online at https://json-schema.org/.

[171] JSONiq: XQuery for JSON. Available online at https://www.w3.org/2011/10/integration-workshop/p/Documentation-0.1-JSONiq-Article-en-US.pdf.

[172] Resource Description Framework (RDF). Available online at https://www.w3.org/RDF/.

[173] SPARQL query language for RDF. Available online at https://www.w3.org/TR/rdf-sparql-query/.

[174] The XML data model. Available online at https://www.w3.org/XML/Datamodel.html.

[175] Extensible Markup Language (XML) 1.0 (Fifth Edition). Available online at https://www.w3.org/TR/xml/.

[176] XQuery 3.1: An XML Query Language. Available online at https://www.w3.org/TR/xquery-31/.

[177] W3C XML Schema Definition Language (XSD). Available online at https://www.w3.org/TR/xmlschema11-1/, 2012.

[178] Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*, 2017.

[179] Amazon Redshift. Available online at https://aws.amazon.com/redshift.

[180] From vendor to in-house: How eBay reimagined its analytics landscape. Available online at https://innovation.ebayinc.com/tech/engineering/from-vendor-to-in-house-how-ebay-reimagined-its-analytics-landscape/.

[181] CoreResearch JSON dataset. Available online at https://core.ac.uk/services/dataset, 2022.

[182] ENELShops RDF dataset. Available online at https://old.datahub.io/dataset/enel-shops, 2022.

[183] Foodista RDF dataset. Available online at https://old.datahub.io/dataset/foodistal, 2022.

[184] GitHub JSON dataset. Available online at https://api.github.com/events, 2022.

[185] Introducing the Knowledge Graph: things, not strings. Available online at https://blog.google/products/search/introducing-knowledge-graph-things-not/.

[186] Cloud data warehouse to power your data-driven innovation. Available online at https://cloud.google.com/bigquery?hl=en.

[187] A dimensional modeling manifesto. Available online at https://www.kimballgroup.com/1997/08/a-dimensional-modeling-manifesto/, 1997.

[188] Microsoft XML inference. Available online at https://learn.microsoft.com/en-us/dotnet/standard/data/xml/inferring-schemas-from-xml-documents, 2021.

[189] Mondial XML dataset. Available online at http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/www/repository.html#mondial, 2022.

[190] NASA flights RDF dataset. Available online at https://old.datahub.io/dataset/data-incubator-nasa, 2022.

[191] OWL 2 Web Ontology Language RDF-Based Semantics. Available online at https://www.w3.org/TR/owl2-rdf-based-semantics/.

[192] Prescription JSON dataset. Available online at https://www.kaggle.com/datasets/roamresearch/prescriptionbasedprediction, 2022.

[193] PubMed biomedical database (XML). Available online at https://www.ncbi.nlm.nih.gov/books/NBK25501/, 2022.

[194] WikiMedia XML dump. Available online at https://dumps.wikimedia.org/frwikinews/20221001/, 2022.

[195] Yelp open dataset: An all-purpose dataset for learning. Available online at https://www.yelp.com/dataset, 2018.

[196] Jianye Yang, Wu Yao, and Wenjie Zhang. Keyword search on large graphs: A survey. *Data Science and Engineering*, 6(2), 2021.

[197] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in relational databases: A survey. *IEEE DEBull.*, 2010.

[198] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. SyntaxSQLNet: syntax tree networks for complex and cross-domain text-to-SQL task. *arXiv preprint arXiv:1810.05237*, 2018.

[199] Gongsheng Yuan, Jiaheng Lu, Zhengtong Yan, and Sai Wu. A survey on mapping semi-structured data and graph data to relational data. *ACM Computing Surveys*, 55(10), 2023.

[200] Elisabeta Zagan and Mirela Danubianu. Cloud data lake: The new trend of data storage. In *HORA*, pages 1–4. IEEE, 2021.

[201] Yi Zhang and Zachary G. Ives. Finding related tables in data lakes for interactive data science. In *SIGMOD*, pages 1951–1966, 2020.

[202] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. JOSIE: overlap set similarity search for finding joinable tables in data lakes. In *SIGMOD*, pages 847–864, 2019.

[203] Julian Ziegler, Peter Reimann, Florian Keller, and Bernhard Mitschang. A graph-based approach to manage CAE data in a data lake. *Procedia CIRP*, 93:496–501, 2020.

[204] Mussab Zneika, Claudio Lucchese, Dan Vodislav, and Dimitris Kotzinos. RDF graph summarization based on approximate patterns. In *ISIP (workshops)*, pages 69–87. Springer, 2016.

[205] Mussab Zneika, Claudio Lucchese, Dan Vodislav, and Dimitris Kotzinos. Summarizing linked data RDF graphs using approximate graph pattern mining. In *ICDT*, 2016.

ECOLE DOCTORALE

Titre: Exploration orientée utilisateur de données semi-structurées

Mots clés: intégration de données, extraction d'information, compréhension de données, modèle Entité-Relation

Résumé: La création, l'utilisation et le partage sans précédent des données à travers le monde contribue à de nouvelles applications et opportunités économiques. Ces données sont souvent larges, hétérogènes en schéma et en modèle, et plus ou moins structurées. Par exemple, les journalistes récoltent des jeux de données de différents acteurs: déclarations d'intérêt des sénateurs français (XML), tweets des personnalités politiques françaises (JSON), base de données d'Offshore leaks (PG) sur les compagnies offshores, dont certaines sont françaises, etc. Dans ce cadre, les journalistes ont cruellement besoin d'outils pour gérer et consolider des sources provenant de différents acteurs, et générer des résultats concrets qu'ils partageront avec leurs collègues ou dans les rédactions. Plus généralement, les utilisateurs (novices ou non) qui doivent trouver, utiliser et/ou partager des jeux de données se trouvent face à un exercice difficile. C'est pourquoi nous proposons de nouvelles méthodes pour appréhender, utiliser et partager des jeux de données semi-structurées, i.e., documents XML et JSON, tableaux CSV, graphes RDF et de propriétés. La motivation principale de ce travail est d'aider les utilisateurs dans leur tâche d'exploration, e.g., comprendre la structure de leurs données, trouver des informations intéressantes dans la masse, pouvoir formuler des requêtes sans grande expertise informatique, recouper plusieurs jeux de données provenant de différents acteurs, etc. Nous proposons une approche unifiée des différents modèles de données, une vue globale que nous pensons nécessaire pour tirer le meilleur de toutes ces données.

Title: User-oriented exploration of semi-structured datasets

Keywords: data integration, information extraction, data understanding, Entity-Relationship model

Abstract: The unprecedented creation, use and share of data around the world has led to new applications and economic opportunities. This data is often large, heterogeneous at a schema and model level, and more or less structured. For instance, data journalists collect data from many different actors: French elected people declarations of interest (XML), tweets of political figures (JSON), OffShore leaks database (PG) about offshore companies, some of which are French, etc. In this frame, this is crucial for journalists to have tools to manage and consolidate sources coming from different actors, and generate tangible results they can share with colleagues and in newsrooms. More generally, users (novice or not) who need to find, use and/or share datasets may have a hard time. This is why we propose novel methods to get acquainted, utilize and share semi-structured datasets, i.e., XML and JSON documents, CSV tables, RDF and property graphs. The main motivation of this work is to help users in their data exploration task, e.g., understand the structure of the data, find interesting nuggets of information, formulate queries without strong IT skills, match several datasets coming from different producers on their common features, etc. We present a unified approach of many semi-structured data models; a global view we think necessary to get the most out of all this data.

Institut Polytechnique de Paris

91120 Palaiseau, France